usenix ASSOCIATION

Proceedings of NSDI '12: 9th USENIX Symposium on Networked Systems Design and Implementation

San Jose, CA, USA April 25–27 2012

conference

proceedings

NSDI '12: 9th USENIX Symposium on Networked Systems Design and Implementation

San Jose, CA, USA
April 25–27, 2012

Sponsored by

usenix ASSOCIATION

in cooperation with ACM SIGCOMM and ACM SIGOPS

ISBN 978-931971-92-8

USENIX Association

Proceedings of NSDI ’12: 9th USENIX Symposium on Networked Systems Design and Implementation

April 25–27, 2012
San Jose, CA, USA

Conference Organizers

External Reviewers

NSDI '12: 9th USENIX Symposium on Networked Systems Design and Implementation
April 25–27, 2012
San Jose, CA, USA

Friday, April 27 (continued)

Message from the Program Co-Chairs

As Program Co-Chairs, we are pleased to welcome you to NSDI ’12. This year’s conference presents the best work in the area of networked systems. Our technical program contains papers spanning leading-edge topics ranging from wireless networking to big data and data center networking.

We received 169 paper submissions, close to the high-water mark of 175 submissions in 2010. All submissions that met the formatting and basic quality standards were reviewed by the Program Committee, and in a small number of cases we used external reviewers to complement the expertise of the PC. Our review process included two offline rounds, offline discussion, and face-to-face discussion at the program committee meeting itself. During the first offline round, all papers received three reviews. We then selected 108 papers for a second round of offline reviewing. Overall, we gathered 721 reviews, with an average load of 28.8 reviews per PC member.

After a set of offline HotCRP discussions, we selected 7 papers for early acceptance and 51 papers for consideration at the PC meeting. 25 of the 27 PC members attended the meeting at MIT on December 13, 2011. At the meeting, the PC selected an additional 23 papers for acceptance, for a total of 30 papers in the technical program. Because of the special role that conferences play in our field, all accepted papers were shepherded by a Program Committee member and, where supported by the shepherd, received extra pages to address reviewers’ comments.

We are grateful to everyone whose hard work made this conference possible. Most of all, we are indebted to all of the authors who submitted their work to this conference. We thank the Program Committee for their dedication, timeliness, and hard work in reviewing papers and participating in the extensive discussions at the PC meeting, as well as for their efforts in the shepherding process. We also thank our external reviewers for lending their expertise on short notice. We extend special thanks to Mary McDavitt for her help in organizing the logistics of the PC meeting at MIT.

We are grateful to the conference sponsors for their support and to the USENIX staff for handling the conference logistics, marketing, and proceedings publication; as always, it is a pleasure to work with them. We relied heavily on the HotCRP reviewing software, and we thank Eddie Kohler for his continuing willingness to support and embellish it. As well, we thank Geoff Voelker for the Banal format checker, which reduced our load in enforcing format compliance. Finally, we thank the NSDI ’12 attendees and future readers of these papers: in the end, it is your participation in our field and interest in the work that makes NSDI, and our community, a success.

Steven Gribble, University of Washington
Dina Katabi, Massachusetts Institute of Technology

CORFU: A Shared Log Design for Flash Clusters

Mahesh Balakrishnan[§], Dahlia Malkhi[§], Vijayan Prabhakaran[§]
Ted Wobber[§], Michael Wei[‡], John D. Davis[§]

[§] Microsoft Research Silicon Valley [‡] University of California, San Diego

Abstract

CORFU[1] organizes a cluster of flash devices as a single, shared log that can be accessed concurrently by multiple clients over the network. The CORFU shared log makes it easy to build distributed applications that require strong consistency at high speeds, such as databases, transactional key-value stores, replicated state machines, and metadata services. CORFU can be viewed as a distributed SSD, providing advantages over conventional SSDs such as distributed wear-leveling, network locality, fault tolerance, incremental scalability and geo-distribution. A single CORFU instance can support up to 200K appends/sec, while reads scale linearly with cluster size. Importantly, CORFU is designed to work directly over network-attached flash devices, slashing cost, power consumption and latency by eliminating storage servers.

1 Introduction

Traditionally, system designers have been forced to choose between performance and safety when building large-scale storage systems. Flash storage has the potential to dramatically alter this trade-off, providing persistence as well as high throughput and low latency. The advent of commodity flash drives creates new opportunities in the data center, enabling new designs that are impractical on disk or RAM infrastructure.

In this paper, we posit that flash storage opens the door to shared log designs within data centers, where hundreds of client machines append to the tail of a single log and read from its body concurrently. A shared log is a powerful and versatile primitive for ensuring strong consistency in the presence of failures and asynchrony. It can play many roles in a distributed system: a consensus engine for consistent replication; a transaction arbitrator [13, 22, 25] for isolation and atomicity; an execution history for replica creation, consistent snapshots, and geo-distribution [16]; and even a primary data store that leverages fast appends on underlying media. Flash is an ideal medium for implementing a scalable shared log, supporting fast, contention-free random reads to the body of the log and fast sequential writes to its tail.

One simple option for implementing a flash-based shared log is to outfit a high-end server with an expensive PCI-e SSD (e.g., Fusion-io [2]), replicating it to handle failures and scale read throughput. However, the resulting log is limited in append throughput by the bandwidth of a single server. In addition, interposing bulky, general-purpose servers between the network and flash can create performance bottlenecks, leaving bandwidth under-utilized, and can also offset the power benefits of flash. In contrast, clusters of small flash units have been shown to be balanced, power-efficient and incrementally scalable [5]. Required is a distributed implementation of a shared log that can operate over such clusters.

Accordingly, we present CORFU, a shared log abstraction implemented over a cluster of flash units. In CORFU, each position in the shared log is mapped to a set of flash pages on different flash units. This map is maintained – consistently and compactly – at the clients. To read a particular position in the shared log, a client uses its local copy of this map to determine a corresponding physical flash page, and then directly issues a read to the flash unit storing that page. To append data, a client first determines the next available position in the shared log – using a sequencer node as an optimization for avoiding contention with other appending clients – and then writes data directly to the set of physical flash pages mapped to that position.

CORFU's client-centric design has two objectives. First, it ensures that the append throughput of the log is not a function of the bandwidth of any single flash unit. Instead, clients can append data to the log as fast as the sequencer can assign them 64-bit tokens, i.e., new positions in the log. Our current user-space sequencer runs at 200K tokens/s; assuming 4KB entries and two-way replication, this is sufficient to saturate the write bandwidth of a cluster of 50 Intel X25-V [1] drives, which in turn can support a million random 4KB reads/sec. Es-

[1]CORFU stands for Clusters of Raw Flash Units, and also for an island near Paxos in Greece.

sentially, CORFU's design decouples ordering from I/O, extracting parallelism from the cluster for appends while providing single-copy semantics for the shared log.

Second, placing functionality at the clients reduces the complexity, cost, latency and power consumption of the flash units. In fact, CORFU can operate over SSDs that are attached directly to the network, eliminating general-purpose storage servers from the critical path. In a parallel effort outside the scope of this paper, we have prototyped a network-attached flash unit on an FPGA platform; when used with the CORFU stack, this custom hardware provides the same throughput as a server-based flash unit while using an order of magnitude less power and providing 33% lower latency on reads. Over a cluster of such flash units, CORFU's logging design acts as a distributed SSD, implementing functionality found inside conventional SSDs – such as wear-leveling – at cluster scale.

To realize these benefits of a client-centric design, CORFU needs to handle failures efficiently. When flash units fail, clients must move consistently to a new map from log positions to flash pages. CORFU achieves this via a reconfiguration mechanism (patterned after Vertical Paxos [17]) capable of restoring availability within tens of milliseconds on drive failures. A challenging failure mode peculiar to a client-centric design involves 'holes' in the log; a client can obtain a log position from the sequencer for an append and then crash without completing the write to that position. To handle such situations, CORFU provides a fast hole-filling primitive that allows other clients to complete an unfinished append (or mark the log position as junk) within a millisecond.

CORFU's target applications are infrastructure layers that use the shared log to implement high-level interfaces. In this paper, we present two such applications. CORFU-Store is a key-value store that supports atomic multi-key puts and gets, fast consistent checkpointing and low-latency geo-distribution; these are properties that are difficult to achieve on conventional partitioned key-value stores. CORFU-SMR is a state machine replication library where replicas propose commands by appending to the log and execute commands by playing the log. In addition to these, we are currently building a database, a virtual disk layer, and a reliable multicast mechanism over CORFU.

We evaluate a CORFU implementation on a cluster of 32 Intel X25-V drives attached to servers, showing that it saturates the aggregate storage bandwidth of the cluster at speeds of 400K 4KB reads per second and nearly 200K 4KB appends per second over the network. We also evaluate CORFU running over an FPGA-based network-attached flash unit, showing that it performs end-to-end reads under 0.5 ms and cross-rack mirrored appends under 1 ms. We show that CORFU is capable of recovering from holes within a millisecond, and from crashed drives within 30 ms. Finally, we show that CORFU-Store provides atomic operations at the speed of the log (40K 10-key multi-gets/s and 20K 10-key multi-puts/s, with 4KB values), and that CORFU-SMR runs at 70K 512-byte commands/s with 10 state machine replicas.

To summarize the contributions of this paper: we propose the first complete design and implementation of a shared log abstraction over a flash cluster. This is also the first distributed, shared log design where maximum append throughput is not a function of any single node's I/O bandwidth. We describe low-latency fault-tolerance mechanisms for recovering from flash unit crashes and holes in the log. We present designs for a strongly consistent key-value store and a state machine replication library that use CORFU. Finally, we evaluate CORFU throughput, latency, fault-tolerance and application performance on a 32-drive cluster of server-attached SSDs as well as an FPGA-based network-attached SSD.

2 Motivation

As stated earlier, the key insight in this paper is that flash storage is an ideal medium for shared log designs. The primary argument is that flash provides fast, contention-free random reads, which enables designs where hundreds of clients can concurrently access a shared log. However, the CORFU design of a shared, distributed log makes sense for other reasons, as well. We first offer a quick primer on the properties of flash storage, and then expand on the rationale for a shared, distributed log.

Flash is read and written in increments of pages (typically of size 4KB). Before a page can be overwritten, it must be erased; erasures can only occur at the granularity of multi-page blocks (of size 256KB). Significantly, flash wears out or ages; as a page is erased and overwritten, it becomes less reliable and is eventually unusable. From a performance standpoint, overwriting a randomly selected flash page requires the surrounding block to undergo an erase operation in the critical path, resulting in poor random write speeds.

At the level of a single drive, these problems are masked from applications by the Flash Translation Layer (FTL) within an SSD. SSDs implement a logical address space over raw flash, mapping logical addresses to physical flash pages. Since flash chips within an SSD cannot be easily replaced, the primary goal of the FTL is wear-leveling: ensuring that all flash chips in the drive age, and expire, in unison. FTLs also speed up random writes by maintaining a free pool of extra, pre-erased blocks; however, sequential writes are still significantly faster. Additionally, the OS can *trim* logical addresses on an SSD, allowing the FTL to reclaim and reuse physical pages.

As a result of these properties, the best data structure for a single flash device is a log; it is always best to write sequentially to flash. One reason is performance; random writes are slower than sequential writes both on raw flash and SSDs (as explained above). Depending on the design of the FTL, random writes can also cause significantly greater wear-out than sequential writes. Accordingly, almost all single-machine filesystems or databases designed for flash storage implement a log-structured design within each device; for example, FAWN [5] organizes each of its individual drives as a log.

CORFU extends this theme by organizing an entire cluster of flash drives as a single log. This log is *distributed* across multiple drives and *shared* by multiple clients. We now explain the rationale for these two design decisions.

The case for a shared log: We stated earlier that a shared log is a powerful building block for distributed applications that require strong consistency. We now expand on this point by describing the different ways in which applications can use a fast, flash-based shared log. By 'shared', we mean that multiple clients can read and append to the log concurrently over the network, regardless of how the log is implemented.

Historically, shared log designs have appeared in a diverse array of systems. QuickSilver [13, 22] and Camelot [25] used shared logs for failure atomicity and node recovery. LBRM [14] uses shared logs for recovery from multicast packet loss. Shared logs are also used in distributed storage systems for consistent remote mirroring [16]. In such systems, CORFU can replace disk-based shared logs, providing higher throughput and lower latency.

However, a flash-based shared log also enables new applications that are infeasible on disk-based infrastructure. For instance, Hyder [7] is a recently proposed high-performance database designed around a flash-based shared log, where servers speculatively execute transactions by appending them to the shared log and then using log order to decide commit/abort status. In fact, Hyder was the original motivating application for CORFU and is currently being implemented over our code base. While the original Hyder paper included a brief design for a flash-based shared log, this was never implemented; later in this paper, we describe how – and why – CORFU departs significantly from the Hyder proposal.

Interestingly, a shared log can also be used as a consensus engine, providing functionality identical to consensus protocols such as Paxos (a fact hinted at by the name of our system). Used in this manner, CORFU provides a fast, fault-tolerant service for imposing and durably storing a total order on events in a distributed system. From this perspective, CORFU can be used as a drop-in replacement for disk-based Paxos implementations, leveraging flash storage to provide better performance.

The case for a distributed log: Existing flash-based storage systems scale capacity and throughput by partitioning data across multiple, independent logs, each of which resides on a single flash drive. In a partitioned system, a total order no longer exists on all updates, making it difficult to support operations such as consistent snapshots and atomic updates across partitions. Strongly consistent operations are usually limited in size and scope to a single partition. The high throughput/size ratio of flash results in smaller drives, exacerbating this problem. Even if all the data involved in an atomic update miraculously resides on the same fine-grained partition, the throughput of updates on that data is limited by the I/O capacity of the primary server of the partition.

Specific to flash storage, other problems arise when a system is partitioned into individual per-SSD logs. In particular, the age distribution of the drives in the cluster is tightly coupled to the observed workload; skewed workloads can age drives at different rates, resulting in unpredictable reliability and performance. For example, a range of key-value pairs can become slower and less reliable than the rest of the key-value store if one of them frequently overwritten. Additionally, striping data across SSDs of different ages can bottleneck performance at the oldest SSD in the stripe. Further, administrative policies may require all drives to be replaced together, in which case even wear-out is preferred. Conversely, specific patterns of uneven wear-out may be preferred, to allow the oldest subset of drives to be replaced periodically.

A distributed log solves these problems by spreading writes across the cluster in controlled fashion, decoupling the age distribution of drives from the observed workload; in effect, it implements distributed wear-leveling. In addition, the entire cluster still functions as a single log; as we show later, this makes it possible to implement strongly consistent operations at cluster scale.

Partitioning is ultimately necessary for achieving scale; we do not contend this point. However, modern systems choose to create very fine-grained partitions, at the level of a single drive or a single array attached to a server. CORFU instead allows an entire cluster to act as a single, coarse-grained partition.

3 Design and Implementation

The setting for CORFU is a data center with a large number of application servers (which we call clients) and a cluster of flash units (see Figure 1). Our goal is to provide applications running on the clients with a shared log abstraction implemented over the flash cluster.

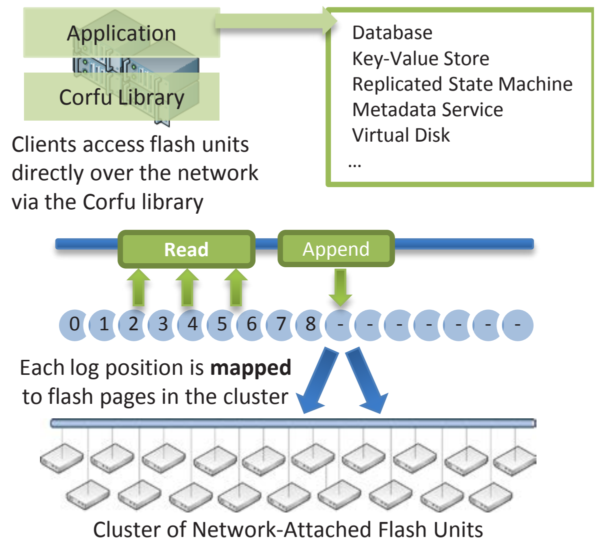

Application
Corfu Library
Database
Key-Value Store
Replicated State Machine
Metadata Service
Virtual Disk
...
Clients access flash units directly over the network via the Corfu library
Read
Append
0 1 2 3 4 5 6 7 8 - - - - - - -
Each log position is mapped to flash pages in the cluster
Cluster of Network-Attached Flash Units

Figure 1: CORFU presents applications running on clients with the abstraction of a shared log, implemented over a cluster of flash units by a client-side library.

Our design for this shared log abstraction is driven by a single imperative: to keep flash units as simple, inexpensive and power-efficient as possible. We achieve this goal by placing all CORFU functionality at the clients and treating flash units as passive storage devices. CORFU clients read and write directly to the address space of each flash unit, coordinating with each other to ensure single-copy semantics for the shared log. Individual flash units do not initiate communication, are unaware of other flash units, and do not participate actively in replication protocols. CORFU does require specific functionality from flash units, which we discuss shortly.

Accordingly, CORFU is implemented as a client-side library that exposes a simple API to applications, shown in Figure 2. The *append* interface adds an entry to the log and returns its position. The *read* interface accepts a position in the log and returns the entry at that position. If no entry exists at that position, an error code is returned. The application can perform garbage collection using *trim*, which indicates to CORFU that no valid data exists at a specific log position. Lastly, the application can *fill* a position with junk, ensuring that it cannot be updated in future with a valid value.

CORFU's task of implementing a shared log abstraction with this API over a cluster of flash units – each of which exposes a separate address space – involves three functions:

- A **mapping function** from logical positions in the log to flash pages on the cluster of flash units.
- A **tail-finding mechanism** for finding the next available logical position on the log for new data.
- A **replication protocol** to write a log entry consistently on multiple flash pages.

append(b)	Append an entry b and return the log position ℓ it occupies
read(ℓ)	Return entry at log position ℓ
trim(ℓ)	Indicate that no valid data exists at log position ℓ
fill(ℓ)	Fill log position ℓ with junk

Figure 2: API exposed by CORFU to applications

These three functions – combined with the ability of clients to read and write directly to the address space of each flash unit – are sufficient to support a shared log abstraction. To read data at a specific log position, the client-side library uses the mapping function to find the appropriate flash page, and then directly issues a read to the device where the flash page is located. To append data, a client finds the tail position of the log, maps it to a set of flash pages, and then initiates the replication protocol that issues writes to the appropriate devices.

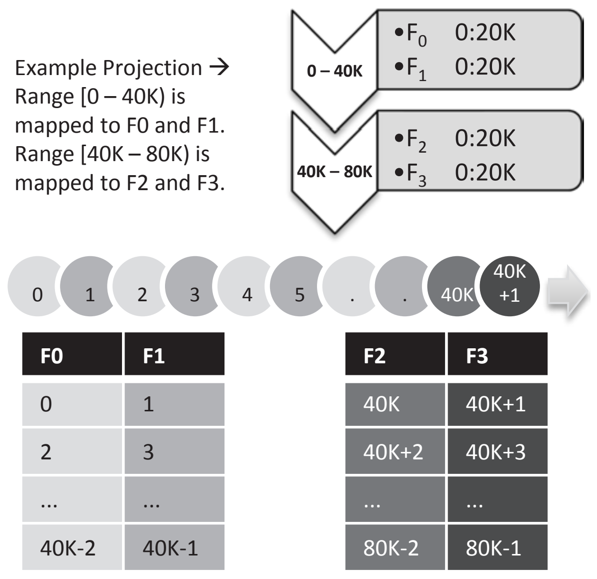
Accordingly, the primary challenges in CORFU revolve around implementing these three functions in an efficient and fault-tolerant manner. Crucially, these functions have to provide single-copy semantics for the shared log even when flash units fail and clients crash.

In this section, we first describe the assumptions made by CORFU about each flash unit. We then describe CORFU's implementation of the three functions described above.

3.1 Flash Unit Requirements

The most basic requirement of a flash unit is that it support reads and writes on an address space of fixed-size pages. We use the term 'flash page' to refer to a page on this address space; however, the flash unit is free to expose a logical address space where logical pages are mapped internally to physical flash pages, as a conventional SSD does. The flash unit is expected to detect and re-map bad blocks in this address space.

To provide single-copy semantics for the shared log, CORFU requires 'write-once' semantics on the flash unit's address space. Reads on pages that have not yet been written should return an error code (*error_unwritten*). Writes on pages that have already been written should also return an error code (*error_overwritten*). In addition to reads and writes, flash units are also required to expose a trim command, allowing clients to indicate that the flash page is not in use anymore.



F0	F1
0	1
2	3
…	…
40K-2	40K-1

F2	F3
40K	40K+1
40K+2	40K+3
…	…
80K-2	80K-1

Figure 3: Example projection that maps different ranges of the shared log onto flash unit extents.

In addition, flash units are required to support a 'seal' command. Each incoming message to a flash unit is tagged with an epoch number. When a particular epoch number is sealed at a flash unit, it must reject all subsequent messages sent with an epoch equal or lower to the sealed epoch. In addition, the flash unit is expected to send back an acknowledgment for the seal command to the sealing entity, including the highest page offset that has been written on its address space thus far.

These requirements – write-once semantics and sealing – are sufficient to ensure CORFU correctness. They are also enough to ensure efficient appends and reads. However, for efficient garbage collection on general-purpose workloads (in terms of network/storage bandwidth and flash erase cycles), CORFU requires that the flash unit expose an infinite address space. We explain this last requirement in detail when we discuss garbage collection and the implementation of flash units.

3.2 Mapping in CORFU

Each CORFU client maintains a local, read-only replica of a data structure called a *projection* that carves the log into disjoint ranges. Each such range is mapped to a list of extents within the address spaces of individual flash units. Figure 3 shows an example projection, where range $[0, 40K)$ is mapped to extents on units F_0 and F_1, while $[40K, 80K)$ is mapped to extents on F_2 and F_3.

Within each range in the log, positions are mapped to flash pages in the corresponding list of extents via a simple, deterministic function. The default function used is round-robin: in the example in Figure 3, log position 0 is mapped to $F_0 : 0$, position 1 is mapped to $F_1 : 0$, position 2 back to $F_0 : 1$, and so on. Any function can be used as long as it is deterministic given a list of extents and a log position. The example above maps each log position to a single flash page; for replication, each extent is associated with a replica set of flash units rather than just one unit. For example, for two-way replication the extent $F_0 : 0 : 20K$ would be replaced by $F_0/F_0' : 0 : 20K$ and the extent $F_1 : 0 : 20K$ would be replaced by $F_1/F_1 : 0 : 20K$.

Accordingly, to map a log position to a set of flash pages, the client first consults its projection to determine the right list of extents for that position; in Figure 3, position $45K$ in the log maps to extents on units F_2 to F_3. It then computes the log position relative to the start of the range; in the example, this is $5K$. Using this relative log position, it applies the deterministic function on the list of extents to determine the flash pages to use. With the round-robin function and the example projection above, the resulting page would be $F_2 : 2500$.

By mapping log positions to flash pages, a projection essentially provides a logical address space implemented over a cluster of flash units. Clients can read or write to positions in this address space by using the projection to determine the flash pages to access. Since CORFU organizes this address space as a log (using a tail-finding mechanism which we describe shortly), clients end up writing only to the last range of positions in the projection ($[40K, 80K)$ in the example); we call this the *active range* in the projection.

3.2.1 Changing the mapping

In a sense, projections are similar to classical views. All operations on flash units – reads, writes, trims – are issued by clients within the context of a single projection. When some event occurs that necessitates a change in the mapping – for example, when a flash unit fails, or when the tail of the log moves past the current active range – a new projection has to be installed on all clients in the system. In effect, each client observes a totally ordered sequence of projections as the position-to-page mapping evolves over time; we call a projection's position in this sequence its epoch. When an operation executes in the context of a projection, all the messages it generates are tagged with the projection's epoch.

As with conventional view change protocols, all participants – in this case, the CORFU clients – must move consistently to a new projection when a change occurs. The new projection should correctly reflect all activity that was successfully completed in any previous projection; i.e., reads must reflect writes and trims that completed in older projections. Further, any activity in-flight during the view change must be aborted and retried in the new projection.

To achieve these properties, CORFU uses a simple, auxiliary-driven *reconfiguration* protocol. The auxiliary is a durably stored sequence of projections in the system, where the position of the projection in the sequence is equivalent to its epoch. Clients can read the ith entry in the auxiliary (getting an error if no such entry exists), or write the ith entry (getting an error if an entry already exists at that position). The auxiliary can be implemented in multiple ways: on a conventional disk volume, as a Paxos state machine, or even as a CORFU instance with a static, never-changing projection.

Auxiliary-driven reconfiguration involves two distinct steps:

1. Sealing the current projection: When a client C_r decides to reconfigure the system from the current projection P_i to a new projection P_{i+1}, it first seals P_i; this involves sending a seal command to a subset of the flash units in P_i. A flash unit in P_i has to be sealed only if a log position mapped to one of its flash pages by P_i is no longer mapped to the same page by P_{i+1}. In practice, this means that only a small subset of flash units have to be sealed, depending on the reason for reconfiguration. Sealing ensures that flash units will reject in-flight messages – writes as well as reads – sent to them in the context of the sealed projection. When clients receive these rejections, they realize that the current projection has been sealed, and wait for a new projection to be installed; if this does not happen, they time out and initiate reconfiguration on their own. The reconfiguring client C_r receives back acknowledgements to the seal command from the flash units, which include the highest offsets written on those flash units thus far. Using this information, it can determine the highest log position reached in the sealed projection; this is useful in certain reconfiguration scenarios.

2. Writing the new projection at the auxiliary: Once the reconfiguring client C_r has successfully sealed the current projection P_i, it attempts to write the new projection P_{i+1} at the $(i+1)th$ position in the auxiliary. If some other client has already written to that position, client C_r aborts its own reconfiguration, reads the existing projection at position $(i+1)$ in the auxiliary, and uses it as its new current projection. As a result, multiple clients can initiate reconfiguration simultaneously, but only one of them succeeds in proposing the new projection.

Projections – and the ability to move consistently between them – offer a versatile mechanism for CORFU to deal with dynamism. Figure 4 shows an example sequence of projections. In Figure 4(A), range $[0, 40K)$ in the log is mapped to the two flash unit mirrored pairs F_0/F_1 and F_2/F_3, while range $[40K, 80K)$ is mapped to F_4/F_5 and F_6/F_7. When F_6 fails with the log tail at position $50K$, CORFU moves to projection (B) immediately, replacing F_6 with F_8 for new appends beyond the current tail of the log, while servicing reads in the log range $[40K, 50K)$ with the remaining mirror F_7.

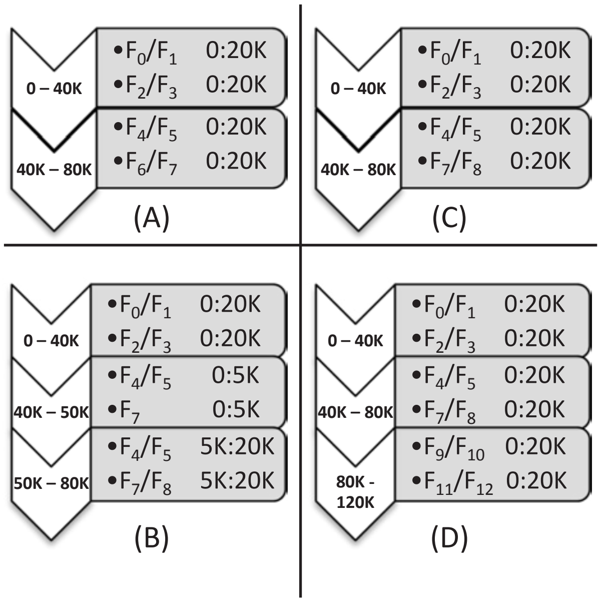


Figure 4: Sequence of projections: When F_6 fails in (A) with the log tail at 50K, clients move to (B) in order to replace F_6 with F_8 for new appends. Once old data on F_6 is rebuilt, F_8 is used in (C) for reads on old data as well. When the log tail goes past $80K$, clients add capacity by moving to (D).

Once F_6 is completely rebuilt on F_8 (by copying entries from F_7), the system moves to projection (C), where F_8 is now used to service all reads in the range $[40K, 80K)$. Eventually, the log tail moves past $80K$, and the system again reconfigures, adding a new range in projection (D) to service reads and writes past $80K$.

3.3 Finding the tail in CORFU

Thus far, we described the machinery used by CORFU to map log positions to sets of flash pages. This allows clients to read or write any position in a logical address space. To treat this address space as an appendable log, clients must be able to find the tail of the log and write to it.

One possible solution is to allow clients to contend for positions. In this case, when a CORFU instance is started, every client that wishes to append data will try to concurrently write to position 0. One client will win, while the rest fail; these clients then try again on position 1, and so on. This approach provides log semantics if only one write is allowed to 'win' on each position;

i.e., complete successfully with the guarantee that any subsequent read on the position returns the value written, until the position is trimmed. We call this property safety-under-contention.

In the absence of replication, this property is satisfied trivially by the flash unit's write-once semantics. When each position is replicated on multiple flash pages, it is still possible to provide this property; in fact, the replication protocol used in CORFU (that we describe next) does so. However, it is clear that such an approach will result in poor performance when there are hundreds of clients concurrently attempting appends to the log.

To eliminate such contention at the tail of the log, CORFU uses a dedicated sequencer that assigns clients 'tokens', corresponding to empty log positions. The sequencer can be thought of as a simple networked counter. To append data, a client first goes to the sequencer, which returns its current value and increments itself. The client has now reserved a position in the log and can write to it without contention from other clients.

Importantly, the sequencer does not represent a single point of failure; it is merely an optimization to reduce contention in the system and is not required for either safety or progress. For fast recovery from sequencer failure, we store the identity of the current sequencer in the projection and use reconfiguration to change sequencers. The counter of the new sequencer is determined using the highest page written on each flash unit, which is returned by the flash unit in response to the seal command during reconfiguration.

However, CORFU's sequencer-based approach does introduce a new failure mode, since 'holes' can appear in the log when clients obtain tokens and then fail to use them immediately due to crashes or slowdowns. Holes can cripple applications that consume the log in strict order, such as state machine replication or transaction processing, since no progress can be made until the status of the hole is resolved. Given that a large system is likely to have a few malfunctioning clients at any given time, holes can severely disrupt application performance. A simple solution is to have other clients fill holes aggressively with a reserved 'junk' value. To prevent aggressive hole-filling from burning up flash cycles and network bandwidth, flash units can be junk-aware, simply updating internal meta-data to mark an address as filled with junk instead of writing an actual value to the flash.

Note that filling holes reintroduces contention for log positions: if a client is merely late in using a reserved token and has not really crashed, it could end up competing with another client trying to fill the position with junk, which is equivalent to two clients concurrently writing to the same position. In other words, the sequencer is an optimization that removes contention for log positions in the common case, but does not eliminate it entirely.

3.4 Replication in CORFU

Once a client reserves a new position in the log via the sequencer, it maps this position to a replica set of flash pages in the cluster using the current projection. At this point, it has to write data at these flash pages over the network. The protocol used to write to the set of flash pages has to provide two properties. First, it has to provide the safety-under-contention property described earlier: when multiple clients write to the replica set for a log position, reading clients should observe a single value. Second, it has to provide durability: written data must be visible to reads only after it has sufficient fault tolerance (i.e., reached $f + 1$ replicas). Given the relatively high cost of flash, we require a solution that tolerates f failures with just $f + 1$ replicas; as a result, data must be visible to reads only after it reaches all replicas.

One approach is to have the client write in parallel to the flash units in the set, and wait for all of them to respond before acknowledging the completion of the append to the application. Unfortunately, when appending clients contend for a log position, different values can be written on different replicas, making it difficult to satisfy the safety-under-contention property. Also, satisfying the durability property with parallel writes requires the reading client to access all replicas to determine if a write has completed or not.

Instead, CORFU uses a simple chaining protocol (essentially, a client-driven variant of Chain Replication [28]) to achieve the safety-under-contention and durability properties. When a client wants to write to a replica set of flash pages, it updates them in a deterministic order, waiting for each flash unit to respond before moving to the next one. The write is successfully completed when the last flash unit in the chain is updated. As a result, if two clients attempt to concurrently update the same replica set of flash pages, one of them will arrive second at the first unit of the chain and receive an *error_overwrite*. This ensures safety-under-contention.

To read from the replica set, clients have two options. If they are unaware of whether the log position was successfully written to or not (for example, when replaying the log after a power failure), they are required to go to the last unit of the chain. If the last unit has not yet been updated, it will return an *error_unwritten*. This ensures the durability property. Alternatively, if the reading client knows already that the log position has been successfully written to (for example, via an out-of-band notification by the writing client), it can go to any replica in the chain for better read performance.

By efficiently handling contention, chained appends allow a client to rapidly fill holes left in the log by other clients that crashed midway through an append. To fill holes, the client starts by checking the first unit of the chain to determine if a valid value exists in the prefix of the chain. If such a value exists, the client walks down the chain to find the first unwritten replica, and then 'completes' the append by copying over the value to the remaining unwritten replicas in chain order. Alternatively, if the first unit of the chain is unwritten, the client writes the junk value to all the replicas in chain order. CORFU exposes this fast hole filling functionality to applications via a *fill* interface. Applications can use this primitive as aggressively as required, depending on their sensitivity to holes in the shared log.

Reconfiguration and Replication: How does this replication protocol interact with the reconfiguration mechanism described in Section 3.2.1? We first define a *chain property*: the replica chain for a log position in a projection has a written prefix storing a single value and an unwritten suffix. A completed write corresponds to a chain with a full-length prefix and a zero-length suffix.

When the system reconfigures from one projection to another, the replica chain for a position can change from one ordered set of flash pages to another. Trivially, the new replica chain is required to satisfy the chain property. Further, we impose two conditions on the transition: if the old chain had a prefix of non-zero length, the new chain must have one as well with the same value. If the old chain had a zero-length suffix, the new chain must have one too. Lastly, to prevent split-brain scenarios, we require that at least one flash unit in the old replica set be sealed in the old projection's epoch.

To meet these requirements, our current implementation follows the protocol described earlier in Figure 4 for flash unit failures. The system first reconfigures to a new chain without the failed replica. It then prepares a new replica by copying over the completed value on each position, filling any holes it encounters in the process. Once the new replica is ready, the system reconfigures again to add it to the end of the replica chain.

Relationship to Paxos: Each chain of flash units in CORFU can be viewed as a single consensus engine, providing (as Paxos does) an ordered sequence of state machine replication (SMR) commands. In conventional SMR implemented using Paxos, the throughput of the system is typically scaled by partitioning the system across multiple instances, effectively splitting the single stream of totally ordered commands into many unrelated streams. Conversely, CORFU scales SMR throughput by **partitioning over time, not space**; the consensus decision on each successive log entry is handled by a different replica chain. Stitching these multiple command streams together is the job of the projection, which is determined by a separate, auxiliary-driven consensus engine, as described earlier. In a separate report, we examine the foundational differences between Paxos and CORFU in more detail [19].

3.5 Garbage Collection

CORFU provides the abstraction of an infinitely growing log to applications. The application does not have to move data around in the address space of the log to free up space. All it is required to do is use the *trim* interface to inform CORFU when individual log positions are no longer in use. As a result, CORFU makes it easy for developers to build applications over the shared log without worrying about garbage collection strategies. An implication of this approach is that as the application appends to the log and trims positions selectively, the address space of the log can become increasingly sparse.

Accordingly, CORFU has to efficiently support a sparse address space for the shared log. The solution is a two-level mapping. As described before, CORFU uses projections to map from a single infinite address space to individual extents on each flash unit. Each flash-unit then maps a sparse 64-bit address space, broken into extents, onto the physical set of pages. The flash unit has to maintain a hash-map from 64-bit addresses to the physical address space of the flash. In fact, this is a relatively minor departure from the functionality implemented by modern SSDs, which often employ page-level maps from a fixed-size logical address space to a slightly larger physical address space.

Crucially, a projection is a range-to-range mapping, and hence it is quite concise. Despite this, it is possible that an adversarial workload can result in bloated projections; for instance, if each range in the projection has a single valid entry that is never trimmed, the mapping for that range has to be retained in the projection for perpetuity.

Even for such adversarial workloads, it is easy to bound the size of the projection tightly by introducing a small amount of proactive data movement across flash units in order to merge consecutive ranges in the projection. For instance, we estimate that adding 0.1% writes to the system can keep the projection under 25 MB on a 1 TB cluster for an adversarial workload. In practice, we do not expect projections to exceed 10s of KBs for conventional workloads; this is borne out by our experience building applications over CORFU. In any case, handling multi-MB projections is not difficult, since they are static data structures that can be indexed efficiently. Additionally, new projections can be written as deltas to the auxiliary.

3.6 Flash Unit Implementations

Flash units can be viewed as conventional SSDs with network interfaces, supporting reads, writes and trims on an address space. Recall that flash units have to meet three major requirements: write-once semantics, a seal capability, and an infinite address space.

Of these, the seal capability is simple: the flash unit maintains an epoch number *cur_sealed_epoch* and rejects all messages tagged with equal or lower epochs, sending back an *error_badepoch* error code. When a seal command arrives with a new epoch to seal, the flash unit first flushes all ongoing operations and then updates its *cur_sealed_epoch*. It then responds to the seal command with *cur_highest_offset*, the highest address written in the address space it exposes; this is required for reconfiguration, as explained previously.

The infinite address space is essentially just a hash-map from 64-bit virtual addresses to the physical address space of the flash. With respect to write-once semantics, an address is considered unwritten if it does not exist in the hash-map. When an address is written to the first time, an entry is created in the hash-map pointing the virtual address to a valid physical address. Each hash-map entry also has a bit indicating whether the address has been trimmed or not, which is set by the *trim* command.

Accordingly, a read succeeds if the address exists in the hash-map and does not have the trim bit set. It returns *error_trimmed* if the trim bit is set, and *error_unwritten* if the address does not exist in the hash-map. A write to an address that exists in the hash-map returns *error_trimmed* if the trim bit is set and *error_overwritten* if it is not. If the address is not in the hash-map, the write succeeds.

To eventually remove trimmed addresses from the hash-map, the flash unit also maintains a watermark before which no unwritten addresses exist, and removes trimmed addresses from the hash-map that are lower than the watermark. Reads and writes to addresses before the watermark that are not in the hash-map return *error-trimmed* immediately. If the address is in the hash-map, reads succeed while writes return *error_overwritten*.

The flash unit also efficiently supports fill operations that write junk by treating such writes differently from first-class writes. Junk writes are initially treated as conventional writes, either succeeding or returning *error_overwritten* or *error_trimmed* as appropriate. However, instead of writing to the flash or SSD, the flash unit points the hash-map entry to a special address reserved for junk; this ensures that flash cycles and capacity is not wasted. Also, once the hash-map is updated, the entry is immediately trimmed in the scope of the same operation. This removes the need for clients to explicitly track and trim junk in the log.

Currently, we have built two flash unit instantiations: *Server+SSD*: this consists of conventional SATA SSDs attached to servers with network interfaces. The server accepts CORFU commands over the network from clients and implements the functionality described above, issuing reads and writes to the fixed-size address space of the SSD as required. *FPGA+SSD:* an FPGA with a SATA SSD attached to it, prototyped using the Beehive [27] architecture on the BEE3 hardware platform [10]. The FPGA includes a network interface to talk to clients and SATA ports to connect to the SSD. A variant under development runs over raw flash chips, implementing FTL functionality on the FPGA itself.

4 CORFU Applications

We are currently prototyping several applications over CORFU. Two such applications that we have implemented are CORFU-Store, a key-value store, and CORFU-SMR, an implementation of State Machine Replication [23].

CORFU-Store: This is a key-value store that supports a number of properties that are difficult to achieve on partitioned stores, including atomic multi-key puts and gets, distributed snapshots, geo-distribution and distributed rollback/replay. In CORFU-Store, a map-service (which can be replicated) maintains a mapping from keys to shared log offsets. To atomically put multiple keys, clients must first append the key-value pairs to the CORFU log, append a commit record and then send the commit record to the map-service. To atomically get multiple keys, clients must query the map-service for the latest key-offset mappings, and then perform CORFU reads on the returned offsets. This protocol ensures linearizability for single-key puts and gets, as well as atomicity for multi-key puts and gets.

CORFU-Store's shared log design makes it easy to take consistent point-in-time snapshots across the entire key space, simply by playing the shared log up to some position. The key-value store can be geo-distributed asynchronously by copying the log, ensuring that the mirror is always at some prior snapshot of the system, with bounded lag. Additionally, CORFU-Store ensures even wear-out across the cluster despite highly skewed write workloads.

CORFU-SMR: Earlier, we noted that CORFU provides the same functionality as Paxos. Accordingly, CORFU is ideal for implementing replicated state machines. Each SMR server simply plays the log forward to receive the next command to execute. It proposes new commands into the state machine by appending them to the log. The ability to do fast queries on the body of the log means that the log entries can also include the data being executed over, obviating the need for each SMR server to store state persistently.

In our current implementation, each SMR server plays the log forward by issuing reads to the log. This approach can stress the shared log: with N SMR servers running at T commands/sec, the CORFU log will see

$N * T$ reads/sec. However, we retained this design since we did not bottleneck at the shared log in our experiments; instead, we were limited by the ability of each SMR server to receive and process commands. In the future, we expect to explore different approaches to playing the log forward, perhaps by having dedicated machines multicast the contents of the log out to all SMR servers.

Other Applications: As mentioned previously, we are implementing the Hyder database [7] over CORFU. One application in the works is a shared block device that exposes a conventional fixed-size address space; this can then be used to expose multiple, independent virtual disks to client VMs [20]. Another application is reliable multicast, where senders append to a single, channel-specific log before multicasting. A receiver can detect lost packets by observing gaps in the sequence of log positions, and retrieve them from the log.

5 Evaluation

We evaluate CORFU on a cluster of 32 Intel X25V drives. Our experiment setup consists of two racks; each rack contains 8 servers (with 2 drives attached to each server) and 11 clients. Each machine has a 1 Gbps link. Together, the two drives on a server provide around 40,000 4KB read IOPS; accessed over the network, each server bottlenecks on the Gigabit link and gives us around 30,000 4KB read IOPS. Each server runs two processes, one per SSD, which act as individual flash units in the distributed system. Currently, the top-of-rack switches of the two racks are connected to a central 10 Gbps switch; our experiments do not generate more than 8 Gbps of inter-rack traffic. We run two client processes on each of the client machines, for a total of 44 client processes.

In all our experiments, we run CORFU with two-way replication, where appends are mirrored on drives in either rack. Reads go from the client to the replica in the local rack. Accordingly, the total read throughput possible on our hardware is equal to 2 GB/sec (16 servers X 1 Gbps each) or 500K/sec 4KB reads. Append throughput is half that number, since appends are mirrored.

Unless otherwise mentioned, our throughput numbers are obtained by running all 44 client processes against the entire cluster of 32 drives. We measure throughput at the clients over a 60-second period during each run.

In addition to this primary deployment of server-attached SSDs, we also provide some results over the prototype FPGA+SSD flash unit. In this case, the FPGA has two SSDs attached to it and emulates a pair of flash units, and a single CORFU client accesses it over the network. The FPGA runs at 1 Gbps or roughly 30K 4KB reads/sec. When running at full speed, it consumes

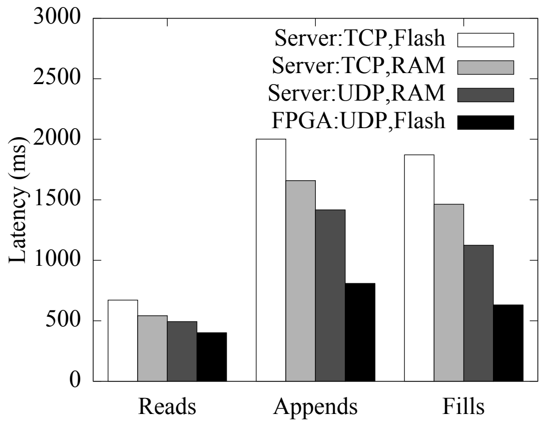


Figure 5: Latency for CORFU operations on different flash unit configurations.

around 15W; in contrast, one of our servers consumes 250W. We also experimented with low-power Atom-based servers, but found them incapable of serving SSD reads over the network at 1 Gbps.

5.1 End-to-end Latency

We first summarize the end-to-end latency characteristics of CORFU in Figure 5. We show the latency for *read*, *append* and *fill* operations issued by clients for four CORFU configurations. The left-most bar for each operation type (Server:TCP,Flash) shows the latency of the server-attached flash unit where clients access the flash unit over TCP/IP when data is durably stored on the SSD; this represents the configuration of our 32-drive deployment. To illustrate the impact of flash latencies on this number, we then show (Server:TCP,RAM), in which the flash unit reads and writes to RAM instead of the SSD. Third, (Server:UDP,RAM) presents the impact of the network stack by replacing TCP with UDP between clients and the flash unit. Lastly, (FPGA:UDP,Flash) shows end-to-end latency for the FPGA+SSD flash unit, with the clients communicating with the unit over UDP.

Against these four configurations we evaluate the latency of three operation types. Reads from the client involve a simple request over the network to the flash unit. Appends involve a token acquisition from the sequencer, and then a chained append over two flash unit replicas. Fills involve an initial read on the head of the chain to check for incomplete appends, and then a chained append to two flash unit replicas.

In this context, Figure 5 makes a number of important points. First, the latency of the FPGA unit is very low for all three operations, providing sub-millisecond appends and fills while satisfying reads within half a millisecond. This justifies our emphasis on a client-centric

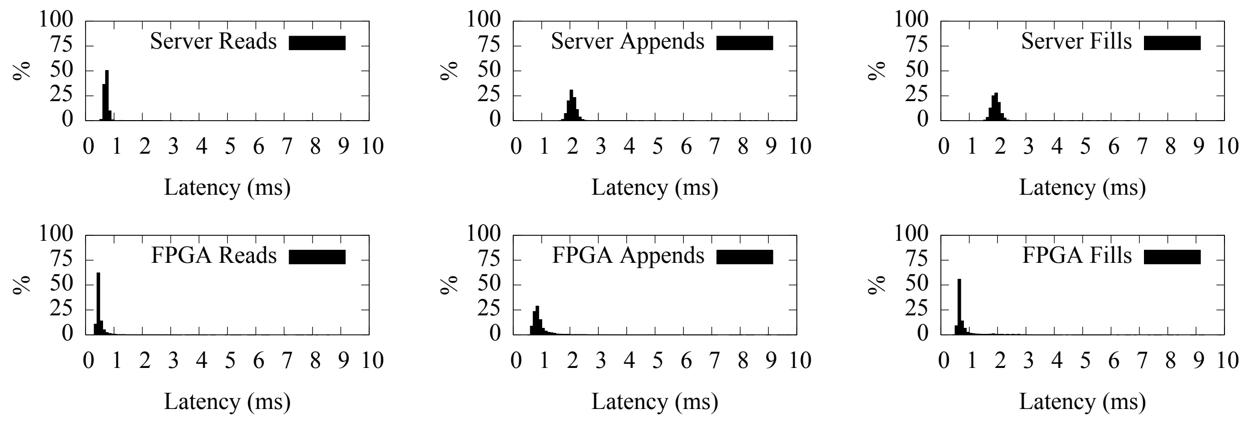


Figure 6: Latency distributions for CORFU operations on 4KB entries.

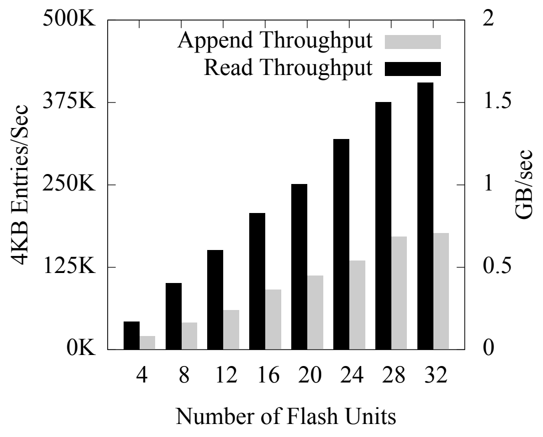


Figure 7: Throughput for random reads and appends.

design; eliminating the server from the critical path appears to have a large impact on latency. Second, the latency to fill a hole in the log is very low; on the FPGA unit, fills complete within 650 microseconds. CORFU's ability to fill holes rapidly is key to realizing the benefits of a client-centric design, since hole-inducing client crashes can be very frequent in large-scale systems. In addition, the chained replication scheme that allows fast fills in CORFU does not impact append latency drastically; on the FPGA unit, appends complete within 750 microseconds.

5.2 Throughput

We now focus on throughput scalability; these experiments are run on our 32-drive cluster of server-attached SSDs. To avoid burning out the SSDs in the throughput experiments, we emulate writes to the SSDs; however, all reads are served from the SSDs, which have 'burnt-in' address spaces that have been written to completely. Emulating SSD writes also allows us to test the CORFU design at speeds exceeding the write bandwidth of our commodity SSDs.

Figure 7 shows how log throughput for 4KB appends and reads scales with the number of drives in the system. As we add drives to the system, both append and read throughput scale up proportionally. Ultimately, append throughput hits a bottleneck at around 180K appends/sec; this is the maximum speed of our sequencer implementation, which is a user-space application running over TCP/IP. Since we were near the append limit of our hardware, we did not further optimize our sequencer implementation.

At such high append throughputs, CORFU can wear out 1 TB of MLC flash in around 4 months. We believe that replacing a $3K cluster of flash drives every four months is acceptable in settings that require strong consistency at high speeds, especially since we see CORFU as a critical component of larger systems (as a consensus engine or a metadata service, for example).

5.3 Reconfiguration

Reconfiguration is used extensively in CORFU, to replace failed drives, to add capacity to the system, and to add, remove, or relocate replicas. This makes reconfiguration latency a crucial metric for our system.

Recall that reconfiguration latency has two components: sealing the current configuration, which contacts a subset of the cluster, and writing the new configuration to the auxiliary. In our experiments, we conservatively seal all drives, to provide an upper bound on reconfiguration time; in practice, only a subset needs to be sealed. Our auxiliary is implemented as a networked file share.

Figure 8 (Left) shows throughput behavior at an appending and reading client when a flash unit fails. When

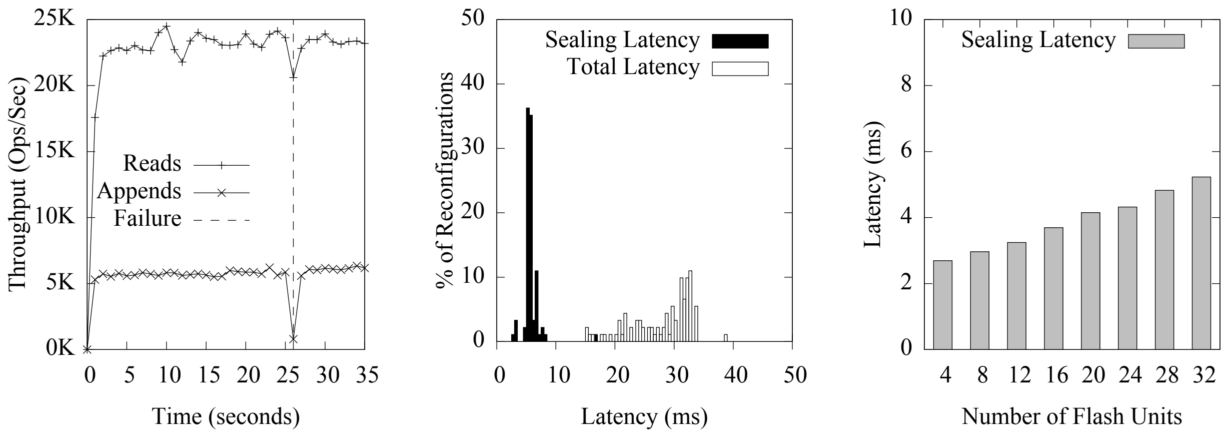


Figure 8: Reconfiguration performance on 32-drive cluster. Left: Appending client waits on failed drive for 1 second before reconfiguring, while reading client continues to read from alive replica. Middle: Distribution of sealing and total reconfiguration latency for 32 drives. Right: Scalability of sealing with number of drives.

the error occurs 25 seconds into the experiment, the appending client's throughput flat-lines and it waits for a 1-second timeout period before reconfiguring to a projection that does not have the failed unit. The reading client, on the other hand, continues reading from the other replica; when reconfiguration occurs, it receives an error from the replica indicating that it has a stale, sealed projection; it then experiences a minor blip in throughput as it retrieves the latest projection from the auxiliary. The graph for sequencer failure looks identical to this graph.

Figure 8 (Middle) shows the distribution of the latency between the start of reconfiguration and its successful completion on a 32-drive cluster. The median latency for the sealing step is around 5 ms, while writing to the auxiliary takes 25 ms more. The slow auxiliary write is due to overheads in serializing the projection as human-readable XML and writing it to the network share; implementing the auxiliary as a static CORFU instance and writing binary data would give us sub-millisecond auxiliary writes (but make the system less easy to administer).

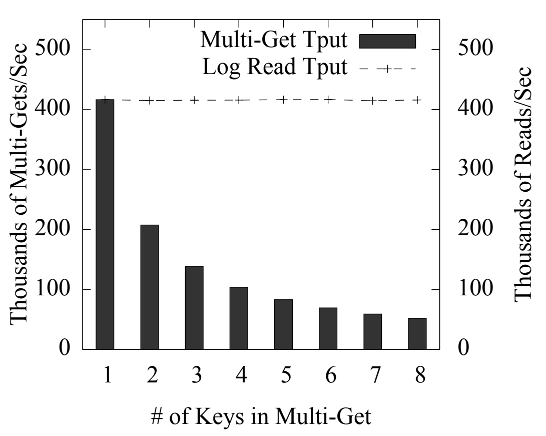
Figure 8 (Right) shows how the median sealing latency scales with the number of drives in the system. Sealing involves the client contacting all flash units in parallel and waiting for responses. The auxiliary write step in reconfiguration is insensitive to system size.

5.4 Applications

We now demonstrate the performance of CORFU-Store for atomic multi-key operations. Figure 9 (Left) shows the performance of multi-put operations in CORFU-Store. On the x-axis, we vary the number of keys updated atomically in each multi-put operation. The bars in the graph plot the number of multi-puts executed per second. The line plots the total number of CORFU log appends resulting as a result of the multi-put operations; a multi-put involving k keys generates $k+1$ log appends, one for each updated key and a final append for the commit record. For small multi-puts involving one or two keys, we are bottlenecked by the ability of the CORFU-Store map-service to handle and process incoming commit records; our current implementation bottlenecks at around 50K single-key commit records per second. As we add more keys to each multi-put, the number of log appends generated increases and the CORFU log bottlenecks at around 180K appends/sec. Beyond 4 keys per multi-put, the log bottleneck determines the number of multi-puts we push through; for example, we obtain $\frac{180K}{6}$=30K multi-puts involving 5 keys.

Figure 9 (Right) shows performance for atomic multi-get operations. As we increase the number of keys accessed by each multi-get, the overall log throughput stays constant while multi-get throughput decreases. For 4 keys per multi-get, we get a little over 100K multi-gets per second, resulting in a load of over 400K reads/s on the CORFU log.

Figure 10 shows the throughput of CORFU-SMR, the state machine replication library implemented over CORFU. In this application, each client acts as a state machine replica, proposing new commands by appending them to the log and executing commands by playing the log. Each command consists of a 512-byte payload and 64 bytes of metadata; accordingly, 7 commands can fit into a single CORFU log entry with batching. In the experiment, each client generates 7K commands/sec; as we add clients on the x-axis, the total rate of commands injected into the system increases by 7K. On the y-axis, we plot the average rate (with error bars signifying the



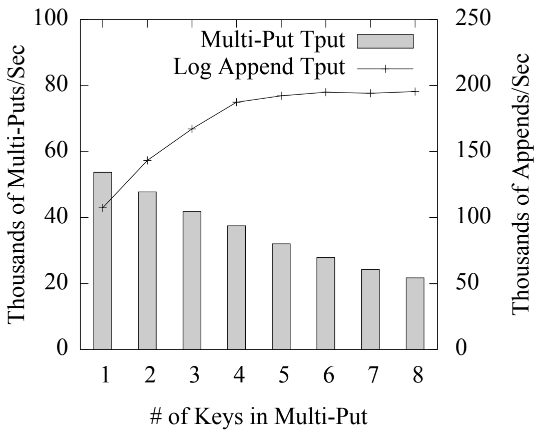


Figure 9: Example CORFU Application: CORFU-Store supports atomic multi-gets and multi-puts at cluster scale.

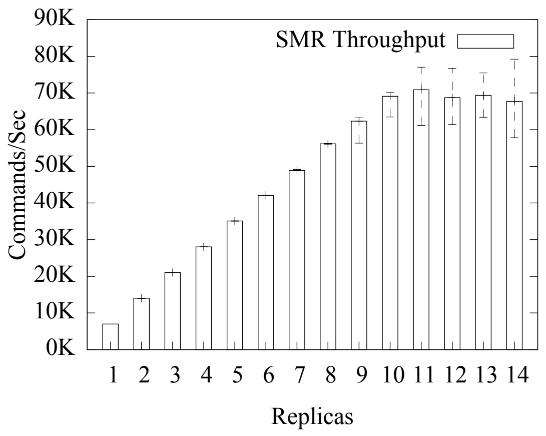


Figure 10: Example CORFU Application: CORFU-SMR supports high-speed state machine replication.

min and max across replicas) at which commands are executed by each replica. While the system has 10 or less CORFU-SMR replicas, the rate at which commands are executed in the system matches the rate at which they are injected. Beyond 10 replicas, we overload the ability of a client to execute incoming commands, and hence throughput flat-lines; this means that clients are now lagging behind while playing the state machine.

6 Related Work

Flash in the Data Center: CORFU's architecture borrows heavily from the FAWN system [5], which first argued for clusters of low-power flash units. CORFU's network-attached flash units are a natural extension of the FAWN argument that low-speed CPUs are more power-efficient; our FPGA implementation consisted of a ring of 100 MHz cores. While FAWN showed that such clusters could efficiently support parallelizable workloads, our contribution lies in extending the benefits of such an architecture to strongly consistent applications.

In contrast to the FAWN and CORFU architecture are PCI-e drives such as Fusion-io [2], which offer up to 10 Gbps of random I/O. Accessing such drives from remote clients requires a high-end, expensive storage server to act as a conduit between the network and the drive. The server is a single point of failure, acts as a bottleneck for reads and writes, is a power hog, costs as much as the drive, and is not incrementally scalable. Storage appliances based on PCI-e drives [3, 4] can solve the first two issues via replication but not the others. CORFU offers similar performance for an order of magnitude less power consumption and less than half the cost (our 32-drive X25-V cluster cost $3K), while offering all the benefits of a distributed solution.

Replicated Data Stores: A traditional design partitions a virtual block drive or a file system onto a collection of independent replica sets. Internally, each object (a file or a block) is replicated using primary-backup mirroring (e.g., [18]) or quorum replication (e.g., [11]). Mapping each object onto a replica set is done by a centralized configuration manager (usually implemented as a replicated state machine). As we explained earlier, CORFU partitions data across time rather than space. In addition, CORFU can act as the configuration manager in such systems, adding a fast source of consistency that other applications can use for resource and lock management. This vital role is is currently played by systems like Chubby [9] and ZooKeeper [15]. Another type of effort related to SMR concentrates on high throughput by allowing reads to proceed from any replica [8, 26]. CORFU faces a similar challenge of learning which appends have committed in order to read from any replica.

Log-structured Filesystems: All contemporary designs for flash storage borrow heavily from a long line of log-structured filesystems starting with LFS [21]. Modern SSDs are testament to the fact that flash is ideally suited for logging designs, eliminating the traditional problem with such systems of garbage collection interfering with foreground activity [24]. Zebra [12] and xFS [6] extended LFS to a distributed setting, striping logs across a collection of storage servers. Like these systems, CORFU stripes updates across drives in a log-structured manner. Unlike them, it implements a single shared log that can be concurrently accessed by multiple clients while providing single-copy semantics.

Comparison with Hyder: As mentioned, Hyder was our original motivating application, and is currently being implemented over CORFU. The Hyder paper [7] included a design outline of a flash-based shared log (we call this Hyder-Log) that was simulated but not implemented. In fact, the design requirements for CORFU emerged from discussions with the Hyder authors on adding fault-tolerance and scalability to their proposal.

CORFU differs from the Hyder-Log design in multiple ways. First, CORFU's use of an off-path sequencer ensures that no single flash unit has to process all append I/O in the system; in contrast, Hyder-Log funnels all appends through a primary unit, and accordingly append throughput is bounded by the speed of a single unit. Second, Hyder-Log parallelizes appends by striping individual entries across units; this forces applications to use large entry sizes that are multiples of 4KB (typical flash page granularity). To achieve parallelism without imposing large entry sizes, CORFU uses projections to map log positions to units in round-robin fashion.

Finally, our experience with CORFU identified holes in the log as the most common mode of failure and source of delays; accordingly, we focused on providing a sub-millisecond hole-filling primitive that ensures high, stable log throughput despite client crashes. Hyder-Log suffers from holes as well; it resorts to a heavy recovery protocol that involves sealing the system and forcing clients to move to a new view every time a hole has to be filled. As a result of these differences, CORFU departs extensively in its design from the Hyder-Log proposal.

7 Conclusion

New storage designs are required to unlock the full potential of flash storage. In this paper, we presented the CORFU system, which organizes a cluster of flash drives as a single, shared log. CORFU offers single-copy semantics at cluster-scale speeds, providing a scalable source of atomicity and durability for distributed systems. CORFU's novel client-centric design eliminates storage servers in favor of simple, efficient and inexpensive flash chips that attach directly to the network.

References

[1] Intel x25-v datasheet. `http://www.intel.com/design/flash/nand/value/technicaldocuments.htm`.

[2] Fusion-io. `http://www.fusionio.com/`, 2011.

[3] Texas memory systems. http://www.ramsan.com/, 2011.

[4] Violin memory. http://www.violin-memory.com/, 2011.

[5] D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan. FAWN: A Fast Array of Wimpy Nodes. In *SOSP 2009*.

[6] T. Anderson, M. Dahlin, J. Neefe, D. Patterson, D. Roselli, and R. Wang. Serverless Network File Systems. In *ACM SIGOPS Operating Systems Review*, volume 29, pages 109–126. ACM, 1995.

[7] P. Bernstein, C. Reid, and S. Das. Hyder – A Transactional Record Manager for Shared Flash. In *CIDR 2011*, pages 9–20, 2011.

[8] W. Bolosky, D. Bradshaw, R. Haagens, N. Kusters, and P. Li. Paxos Replicated State Machines as the Basis of a High-performance Data Store. In *NSDI 2011*.

[9] M. Burrows. The Chubby lock service for loosely-coupled distributed systems. In *OSDI 2006*.

[10] J. Davis, C. P. Thacker, and C. Chang. BEE3: Revitalizing computer architecture research. In *MSR-TR-2009-45*.

[11] S. Frølund, A. Merchant, Y. Saito, S. Spence, and A. Veitch. FAB: Enterprise Storage Systems on a Shoestring. In *HotOS 2003*.

[12] J. Hartman and J. Ousterhout. The zebra striped network file system. *ACM TOCS*, 13(3):274–310, 1995.

[13] R. Haskin, Y. Malachi, and G. Chan. Recovery management in QuickSilver. *ACM TOCS*, 6(1):82–108, 1988.

[14] H. Holbrook, S. Singhal, and D. Cheriton. Log-based receiver-reliable multicast for distributed interactive simulation. *ACM SIGCOMM CCR*, 25(4):328–341, 1995.

[15] P. Hunt, M. Konar, F. Junqueira, and B. Reed. Zookeeper: wait-free coordination for internet-scale systems. In *USENIX ATC*, pages 11–11. USENIX Association, 2010.

[16] M. Ji, A. Veitch, J. Wilkes, et al. Seneca: remote mirroring done write. In *USENIX 2003 Annual Technical Conference*, 2003.

[17] L. Lamport, D. Malkhi, and L. Zhou. Vertical paxos and primary-backup replication. In *PODC 2009*, pages 312–313. ACM, 2009.

[18] E. Lee and C. Thekkath. Petal: Distributed virtual disks. *ACM SIGOPS Operating Systems Review*, 30(5):84–92, 1996.

[19] D. Malkhi, M. Balakrishnan, J. D. Davis, V. Prabhakaran, and T. Wobber. From Paxos to CORFU: A Flash-Speed Shared Log. *ACM SIGOPS Operating Systems Review*, 46(1):47–51, 2012.

[20] D. Meyer, G. Aggarwal, B. Cully, G. Lefebvre, M. Feeley, N. Hutchinson, and A. Warfield. Parallax: Virtual Disks for Virtual Machines. In *Eurosys 2008*.

[21] M. Rosenblum and J. Ousterhout. The design and implementation of a log-structured file system. *ACM TOCS*, 10(1), Feb. 1992.

[22] F. Schmuck and J. Wylie. Experience with transactions in QuickSilver. In *ACM SIGOPS OSR*, volume 25. ACM, 1991.

[23] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, 22(4):299–319, 1990.

[24] M. Seltzer, K. Smith, H. Balakrishnan, J. Chang, S. McMains, and V. Padmanabhan. File system logging versus clustering: A performance comparison. In *USENIX ATC 1995*.

[25] A. Spector, R. Pausch, and G. Bruell. Camelot: A flexible, distributed transaction processing system. In *Compcon Spring'88*.

[26] J. Terrace and M. Freedman. Object storage on CRAQ: High-throughput chain replication for read-mostly workloads. In *Usenix ATC 2009*.

[27] C. P. Thacker. Beehive: A many-core computer for FPGAs. Unpublished Manuscript.

[28] R. van Renesse and F. B. Schneider. Chain replication for supporting high throughput and availability. In *OSDI 2004*.

Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing

Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma,
Murphy McCauley, Michael J. Franklin, Scott Shenker, Ion Stoica
University of California, Berkeley

Abstract

We present Resilient Distributed Datasets (RDDs), a distributed memory abstraction that lets programmers perform in-memory computations on large clusters in a fault-tolerant manner. RDDs are motivated by two types of applications that current computing frameworks handle inefficiently: iterative algorithms and interactive data mining tools. In both cases, keeping data in memory can improve performance by an order of magnitude. To achieve fault tolerance efficiently, RDDs provide a restricted form of shared memory, based on coarse-grained transformations rather than fine-grained updates to shared state. However, we show that RDDs are expressive enough to capture a wide class of computations, including recent specialized programming models for iterative jobs, such as Pregel, and new applications that these models do not capture. We have implemented RDDs in a system called Spark, which we evaluate through a variety of user applications and benchmarks.

1 Introduction

Cluster computing frameworks like MapReduce [10] and Dryad [19] have been widely adopted for large-scale data analytics. These systems let users write parallel computations using a set of high-level operators, without having to worry about work distribution and fault tolerance.

Although current frameworks provide numerous abstractions for accessing a cluster's computational resources, they lack abstractions for leveraging distributed memory. This makes them inefficient for an important class of emerging applications: those that *reuse* intermediate results across multiple computations. Data reuse is common in many *iterative* machine learning and graph algorithms, including PageRank, K-means clustering, and logistic regression. Another compelling use case is *interactive* data mining, where a user runs multiple ad-hoc queries on the same subset of the data. Unfortunately, in most current frameworks, the only way to reuse data between computations (*e.g.,* between two MapReduce jobs) is to write it to an external stable storage system, *e.g.,* a distributed file system. This incurs substantial overheads due to data replication, disk I/O, and serialization, which can dominate application execution times.

Recognizing this problem, researchers have developed specialized frameworks for some applications that require data reuse. For example, Pregel [22] is a system for iterative graph computations that keeps intermediate data in memory, while HaLoop [7] offers an iterative MapReduce interface. However, these frameworks only support specific computation patterns (*e.g.,* looping a series of MapReduce steps), and perform data sharing implicitly for these patterns. They do not provide abstractions for more general reuse, *e.g.,* to let a user load several datasets into memory and run ad-hoc queries across them.

In this paper, we propose a new abstraction called *resilient distributed datasets (RDDs)* that enables efficient data reuse in a broad range of applications. RDDs are fault-tolerant, parallel data structures that let users explicitly persist intermediate results in memory, control their partitioning to optimize data placement, and manipulate them using a rich set of operators.

The main challenge in designing RDDs is defining a programming interface that can provide fault tolerance *efficiently*. Existing abstractions for in-memory storage on clusters, such as distributed shared memory [24], key-value stores [25], databases, and Piccolo [27], offer an interface based on fine-grained updates to mutable state (*e.g.,* cells in a table). With this interface, the only ways to provide fault tolerance are to replicate the data across machines or to log updates across machines. Both approaches are expensive for data-intensive workloads, as they require copying large amounts of data over the cluster network, whose bandwidth is far lower than that of RAM, and they incur substantial storage overhead.

In contrast to these systems, RDDs provide an interface based on *coarse-grained* transformations (*e.g.,* map, filter and join) that apply the same operation to many data items. This allows them to efficiently provide fault tolerance by logging the transformations used to build a dataset (its *lineage*) rather than the actual data.[1] If a partition of an RDD is lost, the RDD has enough information about how it was derived from other RDDs to recompute

[1]Checkpointing the data in some RDDs may be useful when a lineage chain grows large, however, and we discuss how to do it in §5.4.

just that partition. Thus, lost data can be recovered, often quite quickly, without requiring costly replication.

Although an interface based on coarse-grained transformations may at first seem limited, RDDs are a good fit for many parallel applications, because *these applications naturally apply the same operation to multiple data items*. Indeed, we show that RDDs can efficiently express many cluster programming models that have so far been proposed as separate systems, including MapReduce, DryadLINQ, SQL, Pregel and HaLoop, as well as new applications that these systems do not capture, like interactive data mining. The ability of RDDs to accommodate computing needs that were previously met only by introducing new frameworks is, we believe, the most credible evidence of the power of the RDD abstraction.

We have implemented RDDs in a system called Spark, which is being used for research and production applications at UC Berkeley and several companies. Spark provides a convenient language-integrated programming interface similar to DryadLINQ [31] in the Scala programming language [2]. In addition, Spark can be used interactively to query big datasets from the Scala interpreter. We believe that Spark is the first system that allows a general-purpose programming language to be used at interactive speeds for in-memory data mining on clusters.

We evaluate RDDs and Spark through both microbenchmarks and measurements of user applications. We show that Spark is up to 20× faster than Hadoop for iterative applications, speeds up a real-world data analytics report by 40×, and can be used interactively to scan a 1 TB dataset with 5–7s latency. More fundamentally, to illustrate the generality of RDDs, we have implemented the Pregel and HaLoop programming models on top of Spark, including the placement optimizations they employ, as relatively small libraries (200 lines of code each).

This paper begins with an overview of RDDs (§2) and Spark (§3). We then discuss the internal representation of RDDs (§4), our implementation (§5), and experimental results (§6). Finally, we discuss how RDDs capture several existing cluster programming models (§7), survey related work (§8), and conclude.

2 Resilient Distributed Datasets (RDDs)

This section provides an overview of RDDs. We first define RDDs (§2.1) and introduce their programming interface in Spark (§2.2). We then compare RDDs with finer-grained shared memory abstractions (§2.3). Finally, we discuss limitations of the RDD model (§2.4).

2.1 RDD Abstraction

Formally, an RDD is a read-only, partitioned collection of records. RDDs can only be created through deterministic operations on either (1) data in stable storage or (2) other RDDs. We call these operations *transformations* to differentiate them from other operations on RDDs. Examples of transformations include *map*, *filter*, and *join*.[2]

RDDs do not need to be materialized at all times. Instead, an RDD has enough information about how it was derived from other datasets (its *lineage*) to *compute* its partitions from data in stable storage. This is a powerful property: in essence, a program cannot reference an RDD that it cannot reconstruct after a failure.

Finally, users can control two other aspects of RDDs: *persistence* and *partitioning*. Users can indicate which RDDs they will reuse and choose a storage strategy for them (*e.g.,* in-memory storage). They can also ask that an RDD's elements be partitioned across machines based on a key in each record. This is useful for placement optimizations, such as ensuring that two datasets that will be joined together are hash-partitioned in the same way.

2.2 Spark Programming Interface

Spark exposes RDDs through a language-integrated API similar to DryadLINQ [31] and FlumeJava [8], where each dataset is represented as an object and transformations are invoked using methods on these objects.

Programmers start by defining one or more RDDs through transformations on data in stable storage (*e.g., map* and *filter*). They can then use these RDDs in *actions*, which are operations that return a value to the application or export data to a storage system. Examples of actions include *count* (which returns the number of elements in the dataset), *collect* (which returns the elements themselves), and *save* (which outputs the dataset to a storage system). Like DryadLINQ, Spark computes RDDs lazily the first time they are used in an action, so that it can pipeline transformations.

In addition, programmers can call a *persist* method to indicate which RDDs they want to reuse in future operations. Spark keeps persistent RDDs in memory by default, but it can spill them to disk if there is not enough RAM. Users can also request other persistence strategies, such as storing the RDD only on disk or replicating it across machines, through flags to *persist*. Finally, users can set a persistence priority on each RDD to specify which in-memory data should spill to disk first.

2.2.1 Example: Console Log Mining

Suppose that a web service is experiencing errors and an operator wants to search terabytes of logs in the Hadoop filesystem (HDFS) to find the cause. Using Spark, the operator can load just the error messages from the logs into RAM across a set of nodes and query them interactively. She would first type the following Scala code:

[2]Although individual RDDs are immutable, it is possible to implement mutable state by having multiple RDDs to represent multiple versions of a dataset. We made RDDs immutable to make it easier to describe lineage graphs, but it would have been equivalent to have our abstraction be versioned datasets and track versions in lineage graphs.

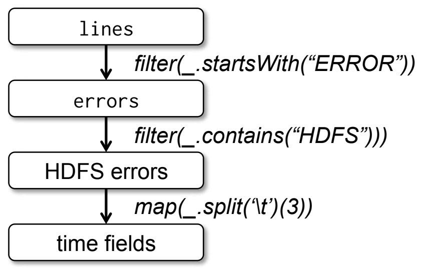


Figure 1: Lineage graph for the third query in our example. Boxes represent RDDs and arrows represent transformations.

```
lines = spark.textFile("hdfs://...")
errors = lines.filter(_.startsWith("ERROR"))
errors.persist()
```

Line 1 defines an RDD backed by an HDFS file (as a collection of lines of text), while line 2 derives a filtered RDD from it. Line 3 then asks for `errors` to persist in memory so that it can be shared across queries. Note that the argument to *filter* is Scala syntax for a closure.

At this point, no work has been performed on the cluster. However, the user can now use the RDD in actions, *e.g.,* to count the number of messages:

```
errors.count()
```

The user can also perform further transformations on the RDD and use their results, as in the following lines:

```
// Count errors mentioning MySQL:
errors.filter(_.contains("MySQL")).count()

// Return the time fields of errors mentioning
// HDFS as an array (assuming time is field
// number 3 in a tab-separated format):
errors.filter(_.contains("HDFS"))
      .map(_.split('\t')(3))
      .collect()
```

After the first action involving `errors` runs, Spark will store the partitions of `errors` in memory, greatly speeding up subsequent computations on it. Note that the base RDD, `lines`, is *not* loaded into RAM. This is desirable because the error messages might only be a small fraction of the data (small enough to fit into memory).

Finally, to illustrate how our model achieves fault tolerance, we show the lineage graph for the RDDs in our third query in Figure 1. In this query, we started with `errors`, the result of a filter on `lines`, and applied a further filter and map before running a *collect*. The Spark scheduler will pipeline the latter two transformations and send a set of tasks to compute them to the nodes holding the cached partitions of `errors`. In addition, if a partition of `errors` is lost, Spark rebuilds it by applying a filter on only the corresponding partition of `lines`.

Aspect	RDDs	Distr. Shared Mem.
Reads	Coarse- or fine-grained	Fine-grained
Writes	Coarse-grained	Fine-grained
Consistency	Trivial (immutable)	Up to app / runtime
Fault recovery	Fine-grained and low-overhead using lineage	Requires checkpoints and program rollback
Straggler mitigation	Possible using backup tasks	Difficult
Work placement	Automatic based on data locality	Up to app (runtimes aim for transparency)
Behavior if not enough RAM	Similar to existing data flow systems	Poor performance (swapping?)

Table 1: Comparison of RDDs with distributed shared memory.

2.3 Advantages of the RDD Model

To understand the benefits of RDDs as a distributed memory abstraction, we compare them against distributed shared memory (DSM) in Table 1. In DSM systems, applications read and write to arbitrary locations in a global address space. Note that under this definition, we include not only traditional shared memory systems [24], but also other systems where applications make fine-grained writes to shared state, including Piccolo [27], which provides a shared DHT, and distributed databases. DSM is a very general abstraction, but this generality makes it harder to implement in an efficient and fault-tolerant manner on commodity clusters.

The main difference between RDDs and DSM is that RDDs can only be created ("written") through coarse-grained transformations, while DSM allows reads and writes to each memory location.[3] This restricts RDDs to applications that perform bulk writes, but allows for more efficient fault tolerance. In particular, RDDs do not need to incur the overhead of checkpointing, as they can be recovered using lineage.[4] Furthermore, only the lost partitions of an RDD need to be recomputed upon failure, and they can be recomputed in parallel on different nodes, without having to roll back the whole program.

A second benefit of RDDs is that their immutable nature lets a system mitigate slow nodes (stragglers) by running backup copies of slow tasks as in MapReduce [10]. Backup tasks would be hard to implement with DSM, as the two copies of a task would access the same memory locations and interfere with each other's updates.

Finally, RDDs provide two other benefits over DSM. First, in bulk operations on RDDs, a runtime can sched-

[3]Note that *reads* on RDDs can still be fine-grained. For example, an application can treat an RDD as a large read-only lookup table.

[4]In some applications, it can still help to checkpoint RDDs with long lineage chains, as we discuss in Section 5.4. However, this can be done in the background because RDDs are immutable, and there is no need to take a snapshot of the *whole* application as in DSM.

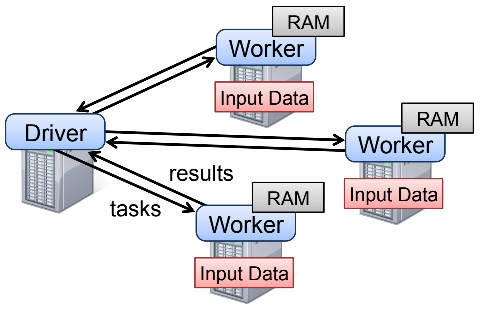


Figure 2: Spark runtime. The user's driver program launches multiple workers, which read data blocks from a distributed file system and can persist computed RDD partitions in memory.

ule tasks based on data locality to improve performance. Second, RDDs degrade gracefully when there is not enough memory to store them, as long as they are only being used in scan-based operations. Partitions that do not fit in RAM can be stored on disk and will provide similar performance to current data-parallel systems.

2.4 Applications Not Suitable for RDDs

As discussed in the Introduction, RDDs are best suited for batch applications that apply the same operation to all elements of a dataset. In these cases, RDDs can efficiently remember each transformation as one step in a lineage graph and can recover lost partitions without having to log large amounts of data. RDDs would be less suitable for applications that make asynchronous fine-grained updates to shared state, such as a storage system for a web application or an incremental web crawler. For these applications, it is more efficient to use systems that perform traditional update logging and data checkpointing, such as databases, RAMCloud [25], Percolator [26] and Piccolo [27]. Our goal is to provide an efficient programming model for batch analytics and leave these asynchronous applications to specialized systems.

3 Spark Programming Interface

Spark provides the RDD abstraction through a language-integrated API similar to DryadLINQ [31] in Scala [2], a statically typed functional programming language for the Java VM. We chose Scala due to its combination of conciseness (which is convenient for interactive use) and efficiency (due to static typing). However, nothing about the RDD abstraction requires a functional language.

To use Spark, developers write a *driver program* that connects to a cluster of *workers*, as shown in Figure 2. The driver defines one or more RDDs and invokes actions on them. Spark code on the driver also tracks the RDDs' lineage. The workers are long-lived processes that can store RDD partitions in RAM across operations.

As we showed in the log mining example in Section 2.2.1, users provide arguments to RDD operations like *map* by passing closures (function literals). Scala represents each closure as a Java object, and these objects can be serialized and loaded on another node to pass the closure across the network. Scala also saves any variables bound in the closure as fields in the Java object. For example, one can write code like `var x = 5; rdd.map(_ + x)` to add 5 to each element of an RDD.[5]

RDDs themselves are statically typed objects parametrized by an element type. For example, RDD[Int] is an RDD of integers. However, most of our examples omit types since Scala supports type inference.

Although our method of exposing RDDs in Scala is conceptually simple, we had to work around issues with Scala's closure objects using reflection [33]. We also needed more work to make Spark usable from the Scala interpreter, as we shall discuss in Section 5.2. Nonetheless, we did *not* have to modify the Scala compiler.

3.1 RDD Operations in Spark

Table 2 lists the main RDD transformations and actions available in Spark. We give the signature of each operation, showing type parameters in square brackets. Recall that *transformations* are lazy operations that define a new RDD, while *actions* launch a computation to return a value to the program or write data to external storage.

Note that some operations, such as *join*, are only available on RDDs of key-value pairs. Also, our function names are chosen to match other APIs in Scala and other functional languages; for example, *map* is a one-to-one mapping, while *flatMap* maps each input value to one or more outputs (similar to the map in MapReduce).

In addition to these operators, users can ask for an RDD to persist. Furthermore, users can get an RDD's partition order, which is represented by a Partitioner class, and partition another dataset according to it. Operations such as *groupByKey*, *reduceByKey* and *sort* automatically result in a hash or range partitioned RDD.

3.2 Example Applications

We complement the data mining example in Section 2.2.1 with two iterative applications: logistic regression and PageRank. The latter also showcases how control of RDDs' partitioning can improve performance.

3.2.1 Logistic Regression

Many machine learning algorithms are iterative in nature because they run iterative optimization procedures, such as gradient descent, to maximize a function. They can thus run much faster by keeping their data in memory.

As an example, the following program implements logistic regression [14], a common classification algorithm

[5]We save each closure at the time it is created, so that the *map* in this example will always add 5 even if `x` changes.

Transformations	*map*(*f* : T ⇒ U)	:	RDD[T] ⇒ RDD[U]
	filter(*f* : T ⇒ Bool)	:	RDD[T] ⇒ RDD[T]
	flatMap(*f* : T ⇒ Seq[U])	:	RDD[T] ⇒ RDD[U]
	sample(*fraction* : Float)	:	RDD[T] ⇒ RDD[T] (Deterministic sampling)
	groupByKey()	:	RDD[(K, V)] ⇒ RDD[(K, Seq[V])]
	reduceByKey(*f* : (V,V) ⇒ V)	:	RDD[(K, V)] ⇒ RDD[(K, V)]
	union()	:	(RDD[T], RDD[T]) ⇒ RDD[T]
	join()	:	(RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (V, W))]
	cogroup()	:	(RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (Seq[V], Seq[W]))]
	crossProduct()	:	(RDD[T], RDD[U]) ⇒ RDD[(T, U)]
	mapValues(*f* : V ⇒ W)	:	RDD[(K, V)] ⇒ RDD[(K, W)] (Preserves partitioning)
	sort(*c* : Comparator[K])	:	RDD[(K, V)] ⇒ RDD[(K, V)]
	partitionBy(*p* : Partitioner[K])	:	RDD[(K, V)] ⇒ RDD[(K, V)]
Actions	*count*()	:	RDD[T] ⇒ Long
	collect()	:	RDD[T] ⇒ Seq[T]
	reduce(*f* : (T,T) ⇒ T)	:	RDD[T] ⇒ T
	lookup(*k* : K)	:	RDD[(K, V)] ⇒ Seq[V] (On hash/range partitioned RDDs)
	save(*path* : String)	:	Outputs RDD to a storage system, *e.g.*, HDFS

Table 2: Transformations and actions available on RDDs in Spark. Seq[T] denotes a sequence of elements of type T.

that searches for a hyperplane *w* that best separates two sets of points (*e.g.,* spam and non-spam emails). The algorithm uses gradient descent: it starts *w* at a random value, and on each iteration, it sums a function of *w* over the data to move *w* in a direction that improves it.

```
val points = spark.textFile(...)
                  .map(parsePoint).persist()
var w = // random initial vector
for (i <- 1 to ITERATIONS) {
  val gradient = points.map{ p =>
    p.x * (1/(1+exp(-p.y*(w dot p.x)))-1)*p.y
  }.reduce((a,b) => a+b)
  w -= gradient
}
```

We start by defining a persistent RDD called `points` as the result of a *map* transformation on a text file that parses each line of text into a Point object. We then repeatedly run *map* and *reduce* on `points` to compute the gradient at each step by summing a function of the current *w*. Keeping `points` in memory across iterations can yield a 20× speedup, as we show in Section 6.1.

3.2.2 PageRank

A more complex pattern of data sharing occurs in PageRank [6]. The algorithm iteratively updates a *rank* for each document by adding up contributions from documents that link to it. On each iteration, each document sends a contribution of $\frac{r}{n}$ to its neighbors, where r is its rank and n is its number of neighbors. It then updates its rank to $\alpha/N + (1-\alpha)\sum c_i$, where the sum is over the contributions it received and N is the total number of documents. We can write PageRank in Spark as follows:

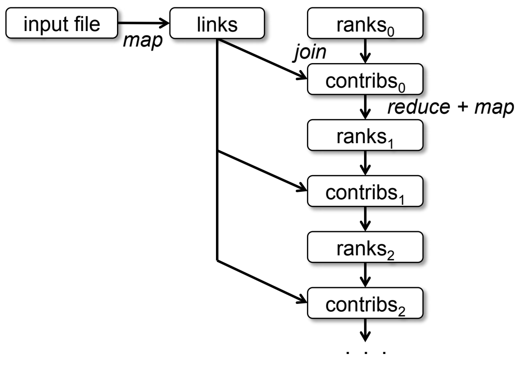


Figure 3: Lineage graph for datasets in PageRank.

```
// Load graph as an RDD of (URL, outlinks) pairs
val links = spark.textFile(...).map(...).persist()
var ranks = // RDD of (URL, rank) pairs
for (i <- 1 to ITERATIONS) {
  // Build an RDD of (targetURL, float) pairs
  // with the contributions sent by each page
  val contribs = links.join(ranks).flatMap {
    (url, (links, rank)) =>
      links.map(dest => (dest, rank/links.size))
  }
  // Sum contributions by URL and get new ranks
  ranks = contribs.reduceByKey((x,y) => x+y)
             .mapValues(sum => a/N + (1-a)*sum)
}
```

This program leads to the RDD lineage graph in Figure 3. On each iteration, we create a new `ranks` dataset based on the `contribs` and `ranks` from the previous iteration and the static `links` dataset.[6] One interesting feature of this graph is that it grows longer with the number

[6]Note that although RDDs are immutable, the variables `ranks` and `contribs` in the program point to different RDDs on each iteration.

of iterations. Thus, in a job with many iterations, it may be necessary to reliably replicate some of the versions of `ranks` to reduce fault recovery times [20]. The user can call *persist* with a `RELIABLE` flag to do this. However, note that the `links` dataset does *not* need to be replicated, because partitions of it can be rebuilt efficiently by rerunning a *map* on blocks of the input file. This dataset will typically be much larger than `ranks`, because each document has many links but only one number as its rank, so recovering it using lineage saves time over systems that checkpoint a program's entire in-memory state.

Finally, we can optimize communication in PageRank by controlling the *partitioning* of the RDDs. If we specify a partitioning for `links` (*e.g.,* hash-partition the link lists by URL across nodes), we can partition `ranks` in the same way and ensure that the *join* operation between `links` and `ranks` requires no communication (as each URL's rank will be on the same machine as its link list). We can also write a custom Partitioner class to group pages that link to each other together (*e.g.,* partition the URLs by domain name). Both optimizations can be expressed by calling `partitionBy` when we define `links`:

```
links = spark.textFile(...).map(...)
             .partitionBy(myPartFunc).persist()
```

After this initial call, the *join* operation between `links` and `ranks` will automatically aggregate the contributions for each URL to the machine that its link lists is on, calculate its new rank there, and join it with its links. This type of consistent partitioning across iterations is one of the main optimizations in specialized frameworks like Pregel. RDDs let the user express this goal directly.

4 Representing RDDs

One of the challenges in providing RDDs as an abstraction is choosing a representation for them that can track lineage across a wide range of transformations. Ideally, a system implementing RDDs should provide as rich a set of transformation operators as possible (*e.g.,* the ones in Table 2), and let users compose them in arbitrary ways. We propose a simple graph-based representation for RDDs that facilitates these goals. We have used this representation in Spark to support a wide range of transformations without adding special logic to the scheduler for each one, which greatly simplified the system design.

In a nutshell, we propose representing each RDD through a common interface that exposes five pieces of information: a set of *partitions*, which are atomic pieces of the dataset; a set of *dependencies* on parent RDDs; a function for computing the dataset based on its parents; and metadata about its partitioning scheme and data placement. For example, an RDD representing an HDFS file has a partition for each block of the file and knows which machines each block is on. Meanwhile, the result of a *map* on this RDD has the same partitions, but applies the map function to the parent's data when computing its elements. We summarize this interface in Table 3.

Operation	Meaning
partitions()	Return a list of Partition objects
preferredLocations(*p*)	List nodes where partition *p* can be accessed faster due to data locality
dependencies()	Return a list of dependencies
iterator(*p, parentIters*)	Compute the elements of partition *p* given iterators for its parent partitions
partitioner()	Return metadata specifying whether the RDD is hash/range partitioned

Table 3: Interface used to represent RDDs in Spark.

The most interesting question in designing this interface is how to represent dependencies between RDDs. We found it both sufficient and useful to classify dependencies into two types: *narrow* dependencies, where each partition of the parent RDD is used by at most one partition of the child RDD, *wide* dependencies, where multiple child partitions may depend on it. For example, *map* leads to a narrow dependency, while *join* leads to to wide dependencies (unless the parents are hash-partitioned). Figure 4 shows other examples.

This distinction is useful for two reasons. First, narrow dependencies allow for pipelined execution on one cluster node, which can compute all the parent partitions. For example, one can apply a *map* followed by a *filter* on an element-by-element basis. In contrast, wide dependencies require data from all parent partitions to be available and to be shuffled across the nodes using a MapReduce-like operation. Second, recovery after a node failure is more efficient with a narrow dependency, as only the lost parent partitions need to be recomputed, and they can be recomputed in parallel on different nodes. In contrast, in a lineage graph with wide dependencies, a single failed node might cause the loss of some partition from all the ancestors of an RDD, requiring a complete re-execution.

This common interface for RDDs made it possible to implement most transformations in Spark in less than 20 lines of code. Indeed, even new Spark users have implemented new transformations (*e.g.,* sampling and various types of joins) without knowing the details of the scheduler. We sketch some RDD implementations below.

HDFS files: The input RDDs in our samples have been files in HDFS. For these RDDs, *partitions* returns one partition for each block of the file (with the block's offset stored in each Partition object), *preferredLocations* gives the nodes the block is on, and *iterator* reads the block.

***map*:** Calling *map* on any RDD returns a MappedRDD object. This object has the same partitions and preferred locations as its parent, but applies the function passed to

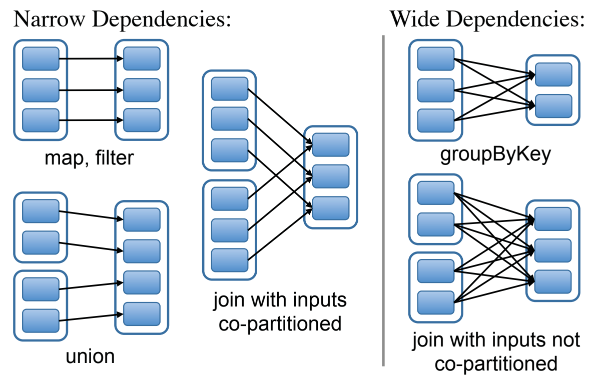

Narrow Dependencies:
Wide Dependencies:
map, filter
union
join with inputs co-partitioned
groupByKey
join with inputs not co-partitioned

Figure 4: Examples of narrow and wide dependencies. Each box is an RDD, with partitions shown as shaded rectangles.

map to the parent's records in its *iterator* method.

***union*:** Calling *union* on two RDDs returns an RDD whose partitions are the union of those of the parents. Each child partition is computed through a narrow dependency on the corresponding parent.[7]

***sample*:** Sampling is similar to mapping, except that the RDD stores a random number generator seed for each partition to deterministically sample parent records.

***join*:** Joining two RDDs may lead to either two narrow dependencies (if they are both hash/range partitioned with the same partitioner), two wide dependencies, or a mix (if one parent has a partitioner and one does not). In either case, the output RDD has a partitioner (either one inherited from the parents or a default hash partitioner).

5 Implementation

We have implemented Spark in about 14,000 lines of Scala. The system runs over the Mesos cluster manager [17], allowing it to share resources with Hadoop, MPI and other applications. Each Spark program runs as a separate Mesos application, with its own driver (master) and workers, and resource sharing between these applications is handled by Mesos.

Spark can read data from any Hadoop input source (*e.g.,* HDFS or HBase) using Hadoop's existing input plugin APIs, and runs on an unmodified version of Scala.

We now sketch several of the technically interesting parts of the system: our job scheduler (§5.1), our Spark interpreter allowing interactive use (§5.2), memory management (§5.3), and support for checkpointing (§5.4).

5.1 Job Scheduling

Spark's scheduler uses our representation of RDDs, described in Section 4.

Overall, our scheduler is similar to Dryad's [19], but it additionally takes into account which partitions of persistent RDDs are available in memory. Whenever a user runs an action (*e.g., count* or *save*) on an RDD, the scheduler examines that RDD's lineage graph to build a DAG of *stages* to execute, as illustrated in Figure 5. Each stage contains as many pipelined transformations with narrow dependencies as possible. The boundaries of the stages are the shuffle operations required for wide dependencies, or any already computed partitions that can short-circuit the computation of a parent RDD. The scheduler then launches tasks to compute missing partitions from each stage until it has computed the target RDD.

[7]Note that our *union* operation does not drop duplicate values.

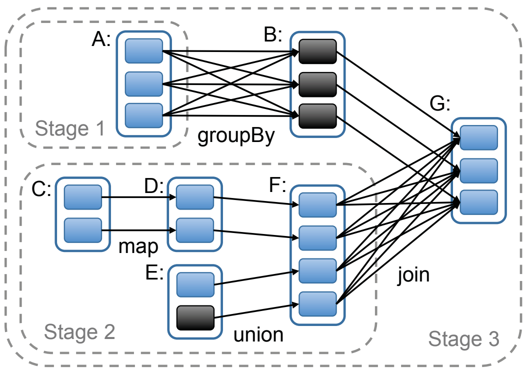


Figure 5: Example of how Spark computes job stages. Boxes with solid outlines are RDDs. Partitions are shaded rectangles, in black if they are already in memory. To run an action on RDD G, we build build stages at wide dependencies and pipeline narrow transformations inside each stage. In this case, stage 1's output RDD is already in RAM, so we run stage 2 and then 3.

Our scheduler assigns tasks to machines based on data locality using delay scheduling [32]. If a task needs to process a partition that is available in memory on a node, we send it to that node. Otherwise, if a task processes a partition for which the containing RDD provides preferred locations (*e.g.,* an HDFS file), we send it to those.

For wide dependencies (*i.e.,* shuffle dependencies), we currently materialize intermediate records on the nodes holding parent partitions to simplify fault recovery, much like MapReduce materializes map outputs.

If a task fails, we re-run it on another node as long as its stage's parents are still available. If some stages have become unavailable (*e.g.,* because an output from the "map side" of a shuffle was lost), we resubmit tasks to compute the missing partitions in parallel. We do not yet tolerate scheduler failures, though replicating the RDD lineage graph would be straightforward.

Finally, although all computations in Spark currently run in response to actions called in the driver program, we are also experimenting with letting tasks on the cluster (*e.g.,* maps) call the *lookup* operation, which provides random access to elements of hash-partitioned RDDs by key. In this case, tasks would need to tell the scheduler to compute the required partition if it is missing.

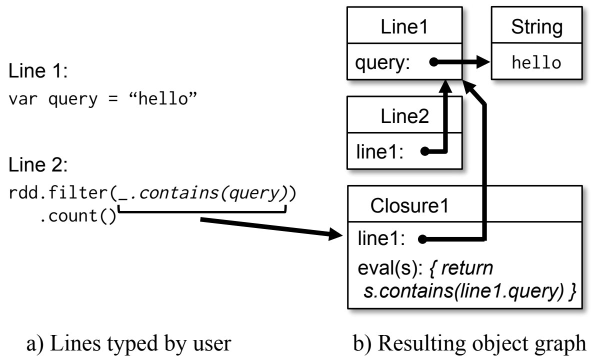


Figure 6: Example showing how the Spark interpreter translates two lines entered by the user into Java objects.

5.2 Interpreter Integration

Scala includes an interactive shell similar to those of Ruby and Python. Given the low latencies attained with in-memory data, we wanted to let users run Spark interactively from the interpreter to query big datasets.

The Scala interpreter normally operates by compiling a class for each line typed by the user, loading it into the JVM, and invoking a function on it. This class includes a singleton object that contains the variables or functions on that line and runs the line's code in an initialize method. For example, if the user types `var x = 5` followed by `println(x)`, the interpreter defines a class called `Line1` containing `x` and causes the second line to compile to `println(Line1.getInstance().x)`.

We made two changes to the interpreter in Spark:

1. *Class shipping:* To let the worker nodes fetch the bytecode for the classes created on each line, we made the interpreter serve these classes over HTTP.
2. *Modified code generation:* Normally, the singleton object created for each line of code is accessed through a static method on its corresponding class. This means that when we serialize a closure referencing a variable defined on a previous line, such as `Line1.x` in the example above, Java will not trace through the object graph to ship the `Line1` instance wrapping around `x`. Therefore, the worker nodes will not receive `x`. We modified the code generation logic to reference the instance of each line object directly.

Figure 6 shows how the interpreter translates a set of lines typed by the user to Java objects after our changes.

We found the Spark interpreter to be useful in processing large traces obtained as part of our research and exploring datasets stored in HDFS. We also plan to use to run higher-level query languages interactively, *e.g.,* SQL.

5.3 Memory Management

Spark provides three options for storage of persistent RDDs: in-memory storage as deserialized Java objects, in-memory storage as serialized data, and on-disk storage. The first option provides the fastest performance, because the Java VM can access each RDD element natively. The second option lets users choose a more memory-efficient representation than Java object graphs when space is limited, at the cost of lower performance.[8] The third option is useful for RDDs that are too large to keep in RAM but costly to recompute on each use.

To manage the limited memory available, we use an LRU eviction policy at the level of RDDs. When a new RDD partition is computed but there is not enough space to store it, we evict a partition from the least recently accessed RDD, unless this is the same RDD as the one with the new partition. In that case, we keep the old partition in memory to prevent cycling partitions from the same RDD in and out. This is important because most operations will run tasks over an entire RDD, so it is quite likely that the partition already in memory will be needed in the future. We found this default policy to work well in all our applications so far, but we also give users further control via a "persistence priority" for each RDD.

Finally, each instance of Spark on a cluster currently has its own separate memory space. In future work, we plan to investigate sharing RDDs across instances of Spark through a unified memory manager.

5.4 Support for Checkpointing

Although lineage can always be used to recover RDDs after a failure, such recovery may be time-consuming for RDDs with long lineage chains. Thus, it can be helpful to checkpoint some RDDs to stable storage.

In general, checkpointing is useful for RDDs with long lineage graphs containing wide dependencies, such as the rank datasets in our PageRank example (§3.2.2). In these cases, a node failure in the cluster may result in the loss of some slice of data from each parent RDD, requiring a full recomputation [20]. In contrast, for RDDs with narrow dependencies on data in stable storage, such as the points in our logistic regression example (§3.2.1) and the link lists in PageRank, checkpointing may never be worthwhile. If a node fails, lost partitions from these RDDs can be recomputed in parallel on other nodes, at a fraction of the cost of replicating the whole RDD.

Spark currently provides an API for checkpointing (a `REPLICATE` flag to *persist*), but leaves the decision of which data to checkpoint to the user. However, we are also investigating how to perform automatic checkpointing. Because our scheduler knows the size of each dataset as well as the time it took to first compute it, it should be able to select an optimal set of RDDs to checkpoint to minimize system recovery time [30].

Finally, note that the read-only nature of RDDs makes

[8]The cost depends on how much computation the application does per byte of data, but can be up to 2× for lightweight processing.

them simpler to checkpoint than general shared memory. Because consistency is not a concern, RDDs can be written out in the background without requiring program pauses or distributed snapshot schemes.

6 Evaluation

We evaluated Spark and RDDs through a series of experiments on Amazon EC2, as well as benchmarks of user applications. Overall, our results show the following:

- Spark outperforms Hadoop by up to 20× in iterative machine learning and graph applications. The speedup comes from avoiding I/O and deserialization costs by storing data in memory as Java objects.
- Applications written by our users perform and scale well. In particular, we used Spark to speed up an analytics report that was running on Hadoop by 40×.
- When nodes fail, Spark can recover quickly by rebuilding only the lost RDD partitions.
- Spark can be used to query a 1 TB dataset interactively with latencies of 5–7 seconds.

We start by presenting benchmarks for iterative machine learning applications (§6.1) and PageRank (§6.2) against Hadoop. We then evaluate fault recovery in Spark (§6.3) and behavior when a dataset does not fit in memory (§6.4). Finally, we discuss results for user applications (§6.5) and interactive data mining (§6.6).

Unless otherwise noted, our tests used `m1.xlarge` EC2 nodes with 4 cores and 15 GB of RAM. We used HDFS for storage, with 256 MB blocks. Before each test, we cleared OS buffer caches to measure IO costs accurately.

6.1 Iterative Machine Learning Applications

We implemented two iterative machine learning applications, logistic regression and k-means, to compare the performance of the following systems:

- *Hadoop:* The Hadoop 0.20.2 stable release.
- *HadoopBinMem:* A Hadoop deployment that converts the input data into a low-overhead binary format in the first iteration to eliminate text parsing in later ones, and stores it in an in-memory HDFS instance.
- *Spark:* Our implementation of RDDs.

We ran both algorithms for 10 iterations on 100 GB datasets using 25–100 machines. The key difference between the two applications is the amount of computation they perform per byte of data. The iteration time of k-means is dominated by computation, while logistic regression is less compute-intensive and thus more sensitive to time spent in deserialization and I/O.

Since typical learning algorithms need tens of iterations to converge, we report times for the first iteration and subsequent iterations separately. We find that sharing data via RDDs greatly speeds up future iterations.

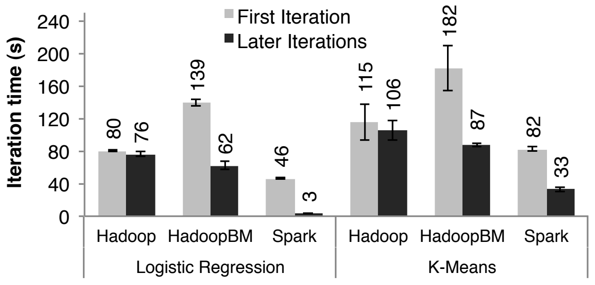


Figure 7: Duration of the first and later iterations in Hadoop, HadoopBinMem and Spark for logistic regression and k-means using 100 GB of data on a 100-node cluster.

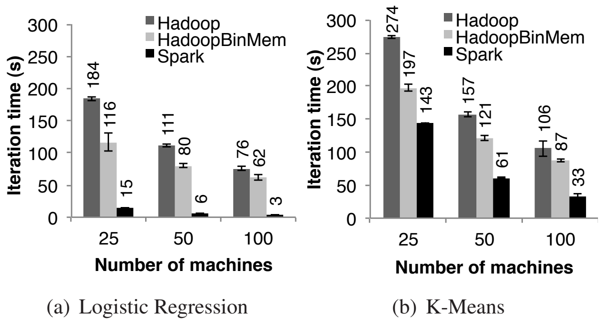


Figure 8: Running times for iterations after the first in Hadoop, HadoopBinMem, and Spark. The jobs all processed 100 GB.

First Iterations All three systems read text input from HDFS in their first iterations. As shown in the light bars in Figure 7, Spark was moderately faster than Hadoop across experiments. This difference was due to signaling overheads in Hadoop's heartbeat protocol between its master and workers. HadoopBinMem was the slowest because it ran an extra MapReduce job to convert the data to binary, it and had to write this data across the network to a replicated in-memory HDFS instance.

Subsequent Iterations Figure 7 also shows the average running times for subsequent iterations, while Figure 8 shows how these scaled with cluster size. For logistic regression, Spark 25.3× and 20.7× faster than Hadoop and HadoopBinMem respectively on 100 machines. For the more compute-intensive k-means application, Spark still achieved speedup of 1.9× to 3.2×.

Understanding the Speedup We were surprised to find that Spark outperformed even Hadoop with in-memory storage of binary data (HadoopBinMem) by a 20× margin. In HadoopBinMem, we had used Hadoop's standard binary format (SequenceFile) and a large block size of 256 MB, and we had forced HDFS's data directory to be on an in-memory file system. However, Hadoop still ran slower due to several factors:

1. Minimum overhead of the Hadoop software stack,
2. Overhead of HDFS while serving data, and

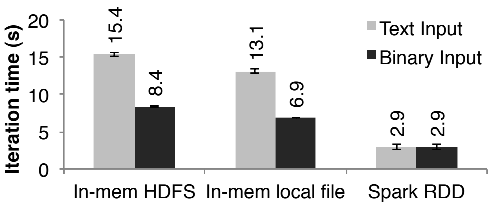


Figure 9: Iteration times for logistic regression using 256 MB data on a single machine for different sources of input.

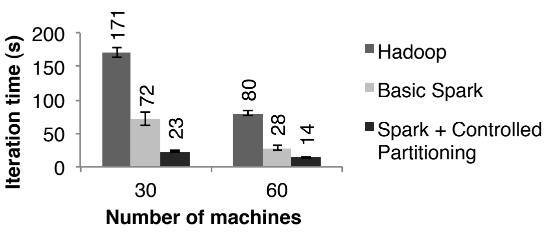


Figure 10: Performance of PageRank on Hadoop and Spark.

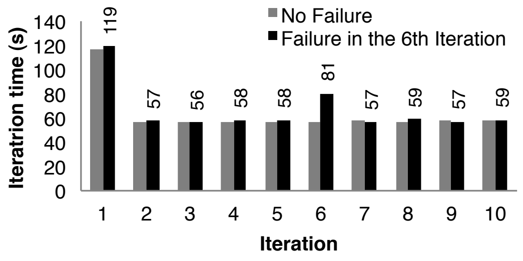


Figure 11: Iteration times for k-means in presence of a failure. One machine was killed at the start of the 6th iteration, resulting in partial reconstruction of an RDD using lineage.

3. Deserialization cost to convert binary records to usable in-memory Java objects.

We investigated each of these factors in turn. To measure (1), we ran no-op Hadoop jobs, and saw that these at incurred least 25s of overhead to complete the minimal requirements of job setup, starting tasks, and cleaning up. Regarding (2), we found that HDFS performed multiple memory copies and a checksum to serve each block.

Finally, to measure (3), we ran microbenchmarks on a single machine to run the logistic regression computation on 256 MB inputs in various formats. In particular, we compared the time to process text and binary inputs from both HDFS (where overheads in the HDFS stack will manifest) and an in-memory local file (where the kernel can very efficiently pass data to the program).

We show the results of these tests in Figure 9. The differences between in-memory HDFS and local file show that reading through HDFS introduced a 2-second overhead, even when data was in memory on the local machine. The differences between the text and binary input indicate the parsing overhead was 7 seconds. Finally, even when reading from an in-memory file, converting the pre-parsed binary data into Java objects took 3 seconds, which is still almost as expensive as the logistic regression itself. By storing RDD elements directly as Java objects in memory, Spark avoids all these overheads.

6.2 PageRank

We compared the performance of Spark with Hadoop for PageRank using a 54 GB Wikipedia dump. We ran 10 iterations of the PageRank algorithm to process a link graph of approximately 4 million articles. Figure 10 demonstrates that in-memory storage alone provided Spark with a 2.4× speedup over Hadoop on 30 nodes. In addition, controlling the partitioning of the RDDs to make it consistent across iterations, as discussed in Section 3.2.2, improved the speedup to 7.4×. The results also scaled nearly linearly to 60 nodes.

We also evaluated a version of PageRank written using our implementation of Pregel over Spark, which we describe in Section 7.1. The iteration times were similar to the ones in Figure 10, but longer by about 4 seconds because Pregel runs an extra operation on each iteration to let the vertices "vote" whether to finish the job.

6.3 Fault Recovery

We evaluated the cost of reconstructing RDD partitions using lineage after a node failure in the k-means application. Figure 11 compares the running times for 10 iterations of k-means on a 75-node cluster in normal operating scenario, with one where a node fails at the start of the 6th iteration. Without any failure, each iteration consisted of 400 tasks working on 100 GB of data.

Until the end of the 5th iteration, the iteration times were about 58 seconds. In the 6th iteration, one of the machines was killed, resulting in the loss of the tasks running on that machine and the RDD partitions stored there. Spark re-ran these tasks in parallel on other machines, where they re-read corresponding input data and reconstructed RDDs via lineage, which increased the iteration time to 80s. Once the lost RDD partitions were reconstructed, the iteration time went back down to 58s.

Note that with a checkpoint-based fault recovery mechanism, recovery would likely require rerunning at least several iterations, depending on the frequency of checkpoints. Furthermore, the system would need to replicate the application's 100 GB working set (the text input data converted into binary) across the network, and would either consume twice the memory of Spark to replicate it in RAM, or would have to wait to write 100 GB to disk. In contrast, the lineage graphs for the RDDs in our examples were all less than 10 KB in size.

6.4 Behavior with Insufficient Memory

So far, we ensured that every machine in the cluster had enough memory to store all the RDDs across itera-

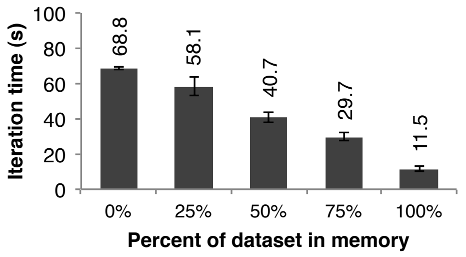


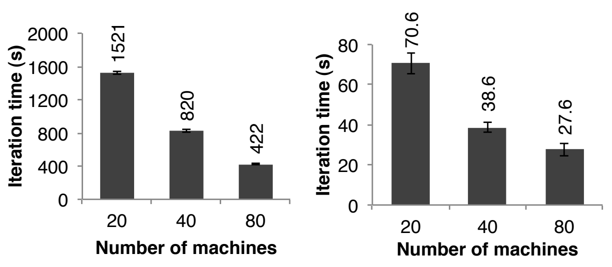
Figure 12: Performance of logistic regression using 100 GB data on 25 machines with varying amounts of data in memory.

tions. A natural question is how Spark runs if there is not enough memory to store a job's data. In this experiment, we configured Spark not to use more than a certain percentage of memory to store RDDs on each machine. We present results for various amounts of storage space for logistic regression in Figure 12. We see that performance degrades gracefully with less space.

6.5 User Applications Built with Spark

In-Memory Analytics Conviva Inc, a video distribution company, used Spark to accelerate a number of data analytics reports that previously ran over Hadoop. For example, one report ran as a series of Hive [1] queries that computed various statistics for a customer. These queries all worked on the same subset of the data (records matching a customer-provided filter), but performed aggregations (averages, percentiles, and `COUNT DISTINCT`) over different grouping fields, requiring separate MapReduce jobs. By implementing the queries in Spark and loading the subset of data shared across them once into an RDD, the company was able to speed up the report by 40×. A report on 200 GB of compressed data that took 20 hours on a Hadoop cluster now runs in 30 minutes using only two Spark machines. Furthermore, the Spark program only required 96 GB of RAM, because it only stored the rows and columns matching the customer's filter in an RDD, not the whole decompressed file.

Traffic Modeling Researchers in the Mobile Millennium project at Berkeley [18] parallelized a learning algorithm for inferring road traffic congestion from sporadic automobile GPS measurements. The source data were a 10,000 link road network for a metropolitan area, as well as 600,000 samples of point-to-point trip times for GPS-equipped automobiles (travel times for each path may include multiple road links). Using a traffic model, the system can estimate the time it takes to travel across individual road links. The researchers trained this model using an expectation maximization (EM) algorithm that repeats two *map* and *reduceByKey* steps iteratively. The application scales nearly linearly from 20 to 80 nodes with 4 cores each, as shown in Figure 13(a).



(a) Traffic modeling (b) Spam classification

Figure 13: Per-iteration running time of two user applications implemented with Spark. Error bars show standard deviations.

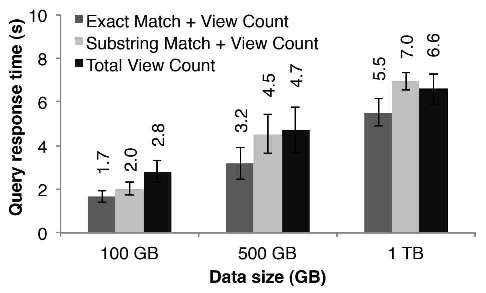


Figure 14: Response times for interactive queries on Spark, scanning increasingly larger input datasets on 100 machines.

Twitter Spam Classification The Monarch project at Berkeley [29] used Spark to identify link spam in Twitter messages. They implemented a logistic regression classifier on top of Spark similar to the example in Section 6.1, but they used a distributed *reduceByKey* to sum the gradient vectors in parallel. In Figure 13(b) we show the scaling results for training a classifier over a 50 GB subset of the data: 250,000 URLs and 10^7 features/dimensions related to the network and content properties of the pages at each URL. The scaling is not as close to linear due to a higher fixed communication cost per iteration.

6.6 Interactive Data Mining

To demonstrate Spark' ability to interactively query big datasets, we used it to analyze 1TB of Wikipedia page view logs (2 years of data). For this experiment, we used 100 `m2.4xlarge` EC2 instances with 8 cores and 68 GB of RAM each. We ran queries to find total views of (1) all pages, (2) pages with titles exactly matching a given word, and (3) pages with titles partially matching a word. Each query scanned the entire input data.

Figure 14 shows the response times of the queries on the full dataset and half and one-tenth of the data. Even at 1 TB of data, queries on Spark took 5–7 seconds. This was more than an order of magnitude faster than working with on-disk data; for example, querying the 1 TB file from disk took 170s. This illustrates that RDDs make Spark a powerful tool for interactive data mining.

7 Discussion

Although RDDs seem to offer a limited programming interface due to their immutable nature and coarse-grained transformations, we have found them suitable for a wide class of applications. In particular, RDDs can express a surprising number of cluster programming models that have so far been proposed as separate frameworks, allowing users to *compose* these models in one program (*e.g.,* run a MapReduce operation to build a graph, then run Pregel on it) and share data between them. In this section, we discuss which programming models RDDs can express and why they are so widely applicable (§7.1). In addition, we discuss another benefit of the lineage information in RDDs that we are pursuing, which is to facilitate debugging across these models (§7.2).

7.1 Expressing Existing Programming Models

RDDs can *efficiently* express a number of cluster programming models that have so far been proposed independently. By "efficiently," we mean that not only can RDDs be used to produce the same output as programs written in these models, but that RDDs can also capture the *optimizations* that these frameworks perform, such as keeping specific data in memory, partitioning it to minimize communication, and recovering from failures efficiently. The models expressible using RDDs include:

MapReduce: This model can be expressed using the *flatMap* and *groupByKey* operations in Spark, or *reduceByKey* if there is a combiner.

DryadLINQ: The DryadLINQ system provides a wider range of operators than MapReduce over the more general Dryad runtime, but these are all bulk operators that correspond directly to RDD transformations available in Spark (*map*, *groupByKey*, *join*, etc).

SQL: Like DryadLINQ expressions, SQL queries perform data-parallel operations on sets of records.

Pregel: Google's Pregel [22] is a specialized model for iterative graph applications that at first looks quite different from the set-oriented programming models in other systems. In Pregel, a program runs as a series of coordinated "supersteps." On each superstep, each vertex in the graph runs a user function that can update state associated with the vertex, change the graph topology, and send messages to other vertices for use in the *next* superstep. This model can express many graph algorithms, including shortest paths, bipartite matching, and PageRank.

The key observation that lets us implement this model with RDDs is that Pregel applies the *same* user function to all the vertices on each iteration. Thus, we can store the vertex states for each iteration in an RDD and perform a bulk transformation (*flatMap*) to apply this function and generate an RDD of messages. We can then join this RDD with the vertex states to perform the message exchange. Equally importantly, RDDs allow us to keep vertex states in memory like Pregel does, to minimize communication by controlling their partitioning, and to support partial recovery on failures. We have implemented Pregel as a 200-line library on top of Spark and refer the reader to [33] for more details.

Iterative MapReduce: Several recently proposed systems, including HaLoop [7] and Twister [11], provide an iterative MapReduce model where the user gives the system a series of MapReduce jobs to loop. The systems keep data partitioned consistently across iterations, and Twister can also keep it in memory. Both optimizations are simple to express with RDDs, and we were able to implement HaLoop as a 200-line library using Spark.

Batched Stream Processing: Researchers have recently proposed several incremental processing systems for applications that periodically update a result with new data [21, 15, 4]. For example, an application updating statistics about ad clicks every 15 minutes should be able to combine intermediate state from the previous 15-minute window with data from new logs. These systems perform bulk operations similar to Dryad, but store application state in distributed filesystems. Placing the intermediate state in RDDs would speed up their processing.

Explaining the Expressivity of RDDs Why are RDDs able to express these diverse programming models? The reason is that the restrictions on RDDs have little impact in many parallel applications. In particular, although RDDs can only be created through bulk transformations, many parallel programs naturally *apply the same operation to many records*, making them easy to express. Similarly, the immutability of RDDs is not an obstacle because one can create multiple RDDs to represent versions of the same dataset. Indeed, many of today's MapReduce applications run over filesystems that do not allow updates to files, such as HDFS.

One final question is why previous frameworks have not offered the same level of generality. We believe that this is because these systems explored specific problems that MapReduce and Dryad do not handle well, such as iteration, without observing that the *common cause* of these problems was a lack of data sharing abstractions.

7.2 Leveraging RDDs for Debugging

While we initially designed RDDs to be deterministically recomputable for fault tolerance, this property also facilitates debugging. In particular, by logging the lineage of RDDs created during a job, one can (1) reconstruct these RDDs later and let the user query them interactively and (2) re-run any task from the job in a single-process debugger, by recomputing the RDD partitions it depends on. Unlike traditional replay debuggers for general distributed systems [13], which must capture or infer the order of events across multiple nodes, this approach adds virtually zero recording overhead because only the RDD lineage graph needs to be logged.[9] We are currently developing a Spark debugger based on these ideas [33].

8 Related Work

Cluster Programming Models: Related work in cluster programming models falls into several classes. First, data flow models such as MapReduce [10], Dryad [19] and Ciel [23] support a rich set of operators for processing data but share it through stable storage systems. RDDs represent a more *efficient* data sharing abstraction than stable storage because they avoid the cost of data replication, I/O and serialization.[10]

Second, several high-level programming interfaces for data flow systems, including DryadLINQ [31] and FlumeJava [8], provide language-integrated APIs where the user manipulates "parallel collections" through operators like *map* and *join*. However, in these systems, the parallel collections represent either files on disk or ephemeral datasets used to express a query plan. Although the systems will pipeline data across operators in the same query (*e.g.,* a *map* followed by another *map*), they cannot share data efficiently *across* queries. We based Spark's API on the parallel collection model due to its convenience, and do not claim novelty for the language-integrated interface, but by providing RDDs as the storage abstraction behind this interface, we allow it to support a far broader class of applications.

A third class of systems provide high-level interfaces for *specific* classes of applications requiring data sharing. For example, Pregel [22] supports iterative graph applications, while Twister [11] and HaLoop [7] are iterative MapReduce runtimes. However, these frameworks perform data sharing implicitly for the pattern of computation they support, and do not provide a general abstraction that the user can employ to share data of her choice among operations of her choice. For example, a user cannot use Pregel or Twister to load a dataset into memory and *then* decide what query to run on it. RDDs provide a distributed storage abstraction explicitly and can thus support applications that these specialized systems do not capture, such as interactive data mining.

Finally, some systems expose shared mutable state to allow the user to perform in-memory computation. For example, Piccolo [27] lets users run parallel functions that read and update cells in a distributed hash table. Distributed shared memory (DSM) systems [24] and key-value stores like RAMCloud [25] offer a similar model. RDDs differ from these systems in two ways. First, RDDs provide a higher-level programming interface based on operators such as *map*, *sort* and *join*, whereas the interface in Piccolo and DSM is just reads and updates to table cells. Second, Piccolo and DSM systems implement recovery through checkpoints and rollback, which is more expensive than the lineage-based strategy of RDDs in many applications. Finally, as discussed in Section 2.3, RDDs also provide other advantages over DSM, such as straggler mitigation.

Caching Systems: Nectar [12] can reuse intermediate results across DryadLINQ jobs by identifying common subexpressions with program analysis [16]. This capability would be compelling to add to an RDD-based system. However, Nectar does not provide in-memory caching (it places the data in a distributed file system), nor does it let users explicitly control which datasets to persist and how to partition them. Ciel [23] and FlumeJava [8] can likewise cache task results but do not provide in-memory caching or explicit control over which data is cached.

Ananthanarayanan et al. have proposed adding an in-memory cache to distributed file systems to exploit the temporal and spatial locality of data access [3]. While this solution provides faster access to data that is already in the file system, it is not as efficient a means of sharing *intermediate* results within an application as RDDs, because it would still require applications to write these results to the file system between stages.

Lineage: Capturing lineage or provenance information for data has long been a research topic in scientific computing and databases, for applications such as explaining results, allowing them to be reproduced by others, and recomputing data if a bug is found in a workflow or if a dataset is lost. We refer the reader to [5] and [9] for surveys of this work. RDDs provide a parallel programming model where fine-grained lineage is inexpensive to capture, so that it can be used for failure recovery.

Our lineage-based recovery mechanism is also similar to the recovery mechanism used *within* a computation (job) in MapReduce and Dryad, which track dependencies among a DAG of tasks. However, in these systems, the lineage information is lost after a job ends, requiring the use of a replicated storage system to share data *across* computations. In contrast, RDDs apply lineage to persist in-memory data efficiently across computations, without the cost of replication and disk I/O.

Relational Databases: RDDs are conceptually similar to views in a database, and persistent RDDs resemble materialized views [28]. However, like DSM systems, databases typically allow fine-grained read-write access to all records, requiring logging of operations and data for fault tolerance and additional overhead to maintain

[9]Unlike these systems, an RDD-based debugger will not replay nondeterministic behavior in the user's functions (*e.g.,* a nondeterministic *map*), but it can at least report it by checksumming data.

[10]Note that running MapReduce/Dryad over an in-memory data store like RAMCloud [25] would still require data replication and serialization, which can be costly for some applications, as shown in §6.1.

consistency. These overheads are not required with the coarse-grained transformation model of RDDs.

9 Conclusion

We have presented resilient distributed datasets (RDDs), an efficient, general-purpose and fault-tolerant abstraction for sharing data in cluster applications. RDDs can express a wide range of parallel applications, including many specialized programming models that have been proposed for iterative computation, and new applications that these models do not capture. Unlike existing storage abstractions for clusters, which require data replication for fault tolerance, RDDs offer an API based on coarse-grained transformations that lets them recover data efficiently using lineage. We have implemented RDDs in a system called Spark that outperforms Hadoop by up to 20× in iterative applications and can be used interactively to query hundreds of gigabytes of data.

We have open sourced Spark at `spark-project.org` as a vehicle for scalable data analysis and systems research.

Acknowledgements

We thank the first Spark users, including Tim Hunter, Lester Mackey, Dilip Joseph, and Jibin Zhan, for trying out our system in their real applications, providing many good suggestions, and identifying a few research challenges along the way. We also thank our shepherd, Ed Nightingale, and our reviewers for their feedback. This research was supported in part by Berkeley AMP Lab sponsors Google, SAP, Amazon Web Services, Cloudera, Huawei, IBM, Intel, Microsoft, NEC, NetApp and VMWare, by DARPA (contract #FA8650-11-C-7136), by a Google PhD Fellowship, and by the Natural Sciences and Engineering Research Council of Canada.

References

[1] Apache Hive. `http://hadoop.apache.org/hive`.

[2] Scala. `http://www.scala-lang.org`.

[3] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica. Disk-locality in datacenter computing considered irrelevant. In *HotOS '11*, 2011.

[4] P. Bhatotia, A. Wieder, R. Rodrigues, U. A. Acar, and R. Pasquin. Incoop: MapReduce for incremental computations. In *ACM SOCC '11*, 2011.

[5] R. Bose and J. Frew. Lineage retrieval for scientific data processing: a survey. *ACM Computing Surveys*, 37:1–28, 2005.

[6] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. In *WWW*, 1998.

[7] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst. HaLoop: efficient iterative data processing on large clusters. *Proc. VLDB Endow.*, 3:285–296, September 2010.

[8] C. Chambers, A. Raniwala, F. Perry, S. Adams, R. R. Henry, R. Bradshaw, and N. Weizenbaum. FlumeJava: easy, efficient data-parallel pipelines. In *PLDI '10*. ACM, 2010.

[9] J. Cheney, L. Chiticariu, and W.-C. Tan. Provenance in databases: Why, how, and where. *Foundations and Trends in Databases*, 1(4):379–474, 2009.

[10] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI*, 2004.

[11] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, and G. Fox. Twister: a runtime for iterative mapreduce. In *HPDC '10*, 2010.

[12] P. K. Gunda, L. Ravindranath, C. A. Thekkath, Y. Yu, and L. Zhuang. Nectar: automatic management of data and computation in datacenters. In *OSDI '10*, 2010.

[13] Z. Guo, X. Wang, J. Tang, X. Liu, Z. Xu, M. Wu, M. F. Kaashoek, and Z. Zhang. R2: an application-level kernel for record and replay. OSDI'08, 2008.

[14] T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer Publishing Company, New York, NY, 2009.

[15] B. He, M. Yang, Z. Guo, R. Chen, B. Su, W. Lin, and L. Zhou. Comet: batched stream processing for data intensive distributed computing. In *SoCC '10*.

[16] A. Heydon, R. Levin, and Y. Yu. Caching function calls using precise dependencies. In *ACM SIGPLAN Notices*, pages 311–320, 2000.

[17] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *NSDI '11*.

[18] T. Hunter, T. Moldovan, M. Zaharia, S. Merzgui, J. Ma, M. J. Franklin, P. Abbeel, and A. M. Bayen. Scaling the Mobile Millennium system in the cloud. In *SOCC '11*, 2011.

[19] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *EuroSys '07*, 2007.

[20] S. Y. Ko, I. Hoque, B. Cho, and I. Gupta. On availability of intermediate data in cloud computations. In *HotOS '09*, 2009.

[21] D. Logothetis, C. Olston, B. Reed, K. C. Webb, and K. Yocum. Stateful bulk processing for incremental analytics. SoCC '10.

[22] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *SIGMOD*, 2010.

[23] D. G. Murray, M. Schwarzkopf, C. Smowton, S. Smith, A. Madhavapeddy, and S. Hand. Ciel: a universal execution engine for distributed data-flow computing. In *NSDI*, 2011.

[24] B. Nitzberg and V. Lo. Distributed shared memory: a survey of issues and algorithms. *Computer*, 24(8):52 –60, Aug 1991.

[25] J. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, S. Mitra, A. Narayanan, G. Parulkar, M. Rosenblum, S. M. Rumble, E. Stratmann, and R. Stutsman. The case for RAMClouds: scalable high-performance storage entirely in DRAM. *SIGOPS Op. Sys. Rev.*, 43:92–105, Jan 2010.

[26] D. Peng and F. Dabek. Large-scale incremental processing using distributed transactions and notifications. In *OSDI 2010*.

[27] R. Power and J. Li. Piccolo: Building fast, distributed programs with partitioned tables. In *Proc. OSDI 2010*, 2010.

[28] R. Ramakrishnan and J. Gehrke. *Database Management Systems*. McGraw-Hill, Inc., 3 edition, 2003.

[29] K. Thomas, C. Grier, J. Ma, V. Paxson, and D. Song. Design and evaluation of a real-time URL spam filtering service. In *IEEE Symposium on Security and Privacy*, 2011.

[30] J. W. Young. A first order approximation to the optimum checkpoint interval. *Commun. ACM*, 17:530–531, Sept 1974.

[31] Y. Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P. K. Gunda, and J. Currey. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *OSDI '08*, 2008.

[32] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling. In *EuroSys '10*, 2010.

[33] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. Technical Report UCB/EECS-2011-82, EECS Department, UC Berkeley, 2011.

Camdoop: Exploiting In-network Aggregation for Big Data Applications

Paolo Costa[†‡] Austin Donnelly[†] Antony Rowstron[†] Greg O'Shea[†]

[†]*Microsoft Research Cambridge* [‡]*Imperial College London*

Abstract

Large companies like Facebook, Google, and Microsoft as well as a number of small and medium enterprises daily process massive amounts of data in batch jobs and in real time applications. This generates high network traffic, which is hard to support using traditional, oversubscribed, network infrastructures. To address this issue, several novel network topologies have been proposed, aiming at increasing the bandwidth available in enterprise clusters.

We observe that in many of the commonly used workloads, data is aggregated during the process and the output size is a fraction of the input size. This motivated us to explore a different point in the design space. Instead of increasing the bandwidth, we focus on decreasing the traffic by pushing aggregation from the edge into the network.

We built Camdoop, a MapReduce-like system running on CamCube, a cluster design that uses a direct-connect network topology with servers directly linked to other servers. Camdoop exploits the property that CamCube servers forward traffic to perform in-network aggregation of data during the shuffle phase. Camdoop supports the same functions used in MapReduce and is compatible with existing MapReduce applications. We demonstrate that, in common cases, Camdoop significantly reduces the network traffic and provides high performance increase over a version of Camdoop running over a switch and against two production systems, Hadoop and Dryad/DryadLINQ.

1 Introduction

"Big Data" generally refers to a heterogeneous class of business applications that operate on large amounts of data. These include traditional batch-oriented jobs such as data mining, building search indices and log collection and analysis [2,5,41], as well as real time stream processing, web search and advertisement selection [3,4,7,17].

To achieve high scalability, these applications usually adopt the *partition/aggregate* model [13]. In this model, which underpins systems like MapReduce [1, 21] and Dryad/DryadLINQ [30,48], there is a large input data set distributed over many servers. Each server processes its share of the data, and generates a local intermediate result. The set of intermediate results contained on all the servers is then aggregated to generate the final result. Often the intermediate data is large so it is divided across multiple servers which perform aggregation on a subset of the data to generate the final result. If there are N servers in the cluster, then using all N servers to perform the aggregation provides the highest parallelism and it is often the default choice. In some cases there is less choice. For instance, selecting the top k items of a collection requires that the final results be generated by a single server. Another example is a distributed user query, which requires the result at a single server to enable low latency responses [7,13].

The aggregation comprises a *shuffle* phase, where the intermediate data is transferred between the servers, and a *reduce* phase where that data is then locally aggregated on the servers. In the common configuration where all servers participate in the reduce phase, the shuffle phase has an all-to-all traffic pattern with $O(N^2)$ flows. This is challenging for current, oversubscribed, data center clusters. A bandwidth oversubscription of 1:x means that the bisection bandwidth of the data center is reduced by a factor x, so the data transfer rate during the shuffle phase is constrained. In current data center clusters, an oversubscription ratio of 1:5 between the rack and the aggregation switch is often the norm [14]. At core routers oversubscription is normally much higher, 1:20 is not unusual [15] and can be as high as 1:240 [26]. If few servers (possibly just one) participate in the reduce phase, the servers' network links are the bottleneck. Further, the small buffers on commodity top-of-rack switches combined with the large number of correlated flows cause TCP throughput collapsing as buffers are overrun (the *incast* problem) [13,44,45].

These issues have led to proposals for new network topologies for data center clusters [11,26,27,40], which aim at increasing the bandwidth available by removing network oversubscription. However, these approaches only partly mitigate the problem. Fate sharing of links means that the full bisection bandwidth cannot be easily achieved [12]. If the number of servers participating in the reduce phase is small, having more bandwidth in the core of the network does not help since the server links are the bottleneck. Finally, non-oversubscribed designs largely increase wiring complexity and overall costs [31].

In this paper, we follow a different approach to increase network performance: we reduce the amount of traffic in the shuffle phase. Systems like MapReduce already exploit the fact that most reduce functions are commutative and associative, and allow aggregation of intermediate data generated at a server using a `combiner` function [21]. It has also been shown that in oversubscribed clusters, performance can be further improved by performing a second stage of partial aggregation at a rack-level [47]. However, in [47] it is shown that at scale performing rack-level aggregation has a small impact on performance. One of the reasons is that the network link of the aggregating server is fair-shared across all the other servers in the rack and becomes a bottleneck.

We have been exploring the benefit of pushing aggregation into the core network, rather than performing it just at the edge. According to [21], the average final output size in Google jobs is 40.3% of the intermediate data set sizes. In the Facebook and Yahoo jobs analyzed in [18], the reduction in size between the intermediate and the output data is even more pronounced: in 81.7% of the Facebook jobs with a reduce phase, the final output data size is only 5.4% of the intermediate data size (resp. 8.2% in 90.5% of the Yahoo jobs). This demonstrates that there is opportunity to aggregate data during the shuffle phase and, hence, to significantly reduce the traffic.

We use a platform called CamCube [9,20], which, rather than using dedicated switches, distributes the switch functionality across the servers. It uses a direct-connect topology with servers directly connected to other servers. It has the property that, at each hop, packets can be intercepted and modified, making it an ideal platform to experiment with moving functionality into the network.

We have implemented *Camdoop*, a MapReduce-like system running on CamCube that supports full on-path aggregation of data streams. Camdoop builds aggregation trees with the sources of the intermediate data as the children and roots at the servers executing the final reduction. A *convergecast* [46] is performed, where all on-path servers aggregate the data (using a combiner function) as it is forwarded towards the root. Depending on how many keys are in common across the servers' intermediate data sets, this reduces network traffic because, at each hop, only a fraction of the data received is forwarded.

This has several benefits beyond simply reducing the data being transferred. It enables the number of reduce tasks to be a function of the expected *output* data size, rather than a function of the *intermediate* data set size, as is normally the case. All servers that forward traffic will effectively participate in the reduce phase, distributing the computational load. This is important in workloads where the output size is often much smaller than the intermediate data size (e.g., in distributed log data aggregation) or where there is a requirement to generate the result at a single server (e.g., in a *top-k* job).

We present a detailed description of Camdoop, and demonstrate that it can provide significant performance gains, up to two orders of magnitude, when associative and commutative reduce functions are used and common keys exist across the intermediate data. We describe how Camdoop handles server failures, and how we ensure that each entry in the intermediate data is included exactly once in the final output in the presence of failures. We also show that even when the reduce function is not associative and commutative, on-path aggregation still provides benefits: Camdoop distributes the sort load across all servers and enables parallelizing the shuffle with the reduce phase, which further reduces the total job time. Finally, Camdoop leverages a custom transport layer that by exploiting application-level knowledge allows a content-based priority scheduling of packets. Even when aggregation is not performed, this helps ensure that reduce tasks are less likely to stall because of lack of data to process.

Camdoop supports the same set of functions originally used in MapReduce [21] and then in Hadoop [1]. It is designed to be a plug-in replacement for Hadoop and be compatible with the existing MapReduce jobs. We chose MapReduce as the programming model due to its wide applicability and popularity. However, our approach could also be extended to other platforms that use a partition/aggregate model, e.g., Dryad [30] or Storm [7].

2 MapReduce

A MapReduce program consists of four functions: `map`, `reduce`, `combiner`, and `partition`. The input data is split into C chunks and, assuming N servers, approximately C/N chunks are stored per server. Usually a chunk is no larger than S MB, e.g. $S = 64$, to increase parallelism and improve performance if tasks need be rerun.

Each job comprises M `map` tasks, where normally $M = C$ and $C \gg N$. These execute locally on each server processing a chunk of input data. Each map task produces an intermediate data set consisting of (key, value) pairs, stored on the server, sorted by key. If the the `reduce` function is associative and commutative, a `combiner` function can be used to locally aggregate the set of intermediate results generated by a map task. The `combiner` function often just coincides with the `reduce` function.

The intermediate data is then processed by R `reduce` tasks. Each task is responsible for a unique range of keys. For each key, it takes the set of all values associated with that key and outputs a (key, values) pair. Each server can run one or more reduce tasks. The reduce tasks require all the intermediate data consisting of all keys (with the associated values) they are responsible for. Hence, each server

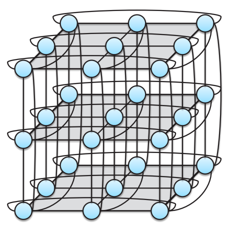


Figure 1: A 3-ary 3-cube (or 3D torus).

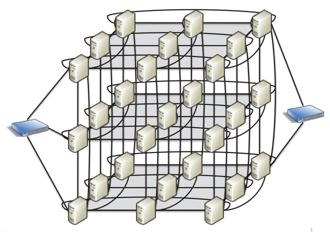


Figure 2: The CamCube network topology.

takes the sorted local intermediate data and splits it into R blocks, with each block containing all keys in a range required by a single reduce task. The blocks are transferred to the server running the reduce task responsible in the *shuffle phase*. If $R \geq N$ then an all-to-all traffic pattern is generated, with each server sending $1/N$th of the locally stored intermediate data to $N-1$ servers.

The `partition` function encodes how the key ranges are split across the reduce tasks. Often, just a default hashing function is used. However, if some additional properties are required (e.g., ensuring that concatenating the output files produced by the reduce tasks generates a correct total ordering of keys), a more complex `partition` function can be used.

The parameter M is simply a function of the input data size and the number of servers N. However, setting the value R is a little more complex and it can have significant performance impact. R has to be set as a function of the *intermediate* data set size, because $1/R$th of the intermediate data set is processed by each reduce task. Obviously, the input data set size is fixed and, provided all the functions are deterministic, the output data set size will simply be a function of the input data set size. However, the intermediate data set size is a function of the input data set size, N, M, and the ratio of input data to output data for the `map` and `combiner` functions.

In some scenarios, for example in web search queries or *top-k* jobs, there is a need to have $R = 1$. In others, for example in multi-round MapReduce jobs, storing the output on a smaller set of servers can be beneficial. Camdoop enables selecting R based on the output data set size, but still allows *all* servers to contribute resources to perform the aggregation, even when $R = 1$.

3 CamCube overview

CamCube [9, 20] is a prototype cluster designed using commodity hardware to experiment with alternative approaches to implementing services running in data centers. CamCube uses a direct-connect topology, in which servers are directly connected to each other using 1 Gbps Ethernet cross-over cables, creating a 3D torus [37] (also known as a k-ary 3-cube), like the one shown in Figure 1. This topology, popular in high performance computing, e.g. IBM BlueGene/L and Cray XT3/Red Storm, provides multiple paths between any source and destination, making it resilient to both link and server failure, and efficient wiring using only short cables is possible [10]. Figure 2 shows an example of a CamCube with 27 servers. Servers are responsible for routing all the intra-CamCube traffic through the direct-connect network. Switches are only used to connect CamCube servers to the external networks but are not used to route internal traffic. Therefore, not all servers need be connected to the switch-based network, as shown in Figure 2.

CamCube uses a novel networking stack, which supports a key-based routing functionality, inspired by the one used in structured overlays. Each server is automatically assigned a 3D coordinate (x,y,z), which is used as the address of the server. This defines a 3D coordinate space similar to the one used in the Content-addressable Network (CAN) [38], and the key-space management service exploits the fact that the physical topology is the same as the virtual topology. In CamCube, keys are encoded using 160-bit identifiers and each server is responsible for a subset of them. Keys are assigned to servers as follows. The most significant k bits are used to generate an (x,y,z) coordinate. If the server with this coordinate is reachable, then the identifier is mapped to this server. If the server is unreachable, the identifier is mapped to one of its neighbors. The remaining $(160-k)$ bits are used to determine the coordinate of this neighbor, or of another server if all neighbors also failed. This deterministic mapping is consistent across all servers, and handles cascading failures.

The main benefit of CamCube is that by using a direct-connect topology and letting servers handle packet forwarding, it completely removes the distinction between the logical and the physical network. This enables services to easily implement custom-routing protocols (e.g., multicast or anycast) as well as efficient in-network services (e.g., caching or in-network aggregation), without incurring the typical overhead (path stretch, link sharing, etc.) and development complexity introduced by overlays.

The CamCube software stack comprises a kernel driver to send and receive raw Ethernet frames. In our prototype, the packets are transferred to a user-space runtime, written in C#. The runtime implements the CamCube API, which provides low-level functionality to send and receive packets to/from the six one-hop physical neighbors. All further functionality, such as key-based routing and failure detection as well as higher-level services (e.g., a key-value store or a graph-processing engine) are implemented in user-space on top of the runtime. The runtime manages the transmission of packets and each service is provided with a service-specific queue for outbound packets. The runtime polls these queues in a round-robin fashion, achieving fair sharing of the outbound links across the services.

Experiments with our prototype show that the overhead of distributing the routing across the servers and of making them participate in packet forwarding is tolerable: servers are able to sustain 12 Gpbs of throughput (i.e., all 6 links running at full capacity) using 21% of the CPU [20].

To fully achieve the performance benefits of CamCube, services should be designed so as to exploit the network topology and server forwarding. For instance, Camdoop obtains high performance by adopting a custom routing and transport protocol, and performing in-network aggregation. Yet, legacy applications can still benefit from CamCube due to its high bisection bandwidth. We implemented a TCP/IP service, which enables running *unmodified* TCP/IP applications on CamCube. To evaluate the performance that these applications could achieve in CamCube, we ran an experiment on our 27-server testbed (described in Section 5) in which every server simultaneously transferred 1 GB of data to each of the other 26 servers using TCP. This creates an all-to-all traffic pattern. We obtained a median aggregate TCP inbound throughput of 1.49 Gbps per server, which is higher than the maximum throughput achievable in a conventional cluster where servers have 1 Gbps uplinks to the switch. We also measured the average RTT, using a simple ping service, obtaining a value of 0.13 ms per each hop. In an 8x8x8 CamCube (512 servers), this would lead to a *worst-case* RTT of 1.56 ms (the average-case would be 0.78 ms).

4 Camdoop

Camdoop is a CamCube service to run MapReduce-like jobs. It exploits the ability of custom forwarding and processing of packets on path to perform in-network aggregation, which improves the performance of the shuffle and reduce phase. An instance of the Camdoop service runs on all CamCube servers. Any CamCube server connected to the switch-based network can act as a front-end and receive jobs from servers external to the CamCube. A job description includes a *jobId*, the code for map, reduce and any other functions required, such as the combiner and partition function, as well as a description of the input data and other runtime data such as the value of R. Camdoop assumes that all the functions are deterministic, which is the normal case in MapReduce jobs. The input data set is stored in a distributed file system implemented on top of a key-value store running on CamCube, as with GFS [25] or HDFS [1]. It is split into multiple chunks, each with a 160-bit identifier (*chunkId*). When a data set is inserted, the chunkIds are generated to ensure that chunks are, approximately, uniformly distributed. The chunkId determines the servers that will store a file replica (by default a chunk is replicated three times). The final output can be written to local disk, inserted into the distributed file system or inserted as key-value pairs into a key-value store. If the distributed file system is used, at job submission, the identifiers of the chunks comprising the output file can be generated so the server running the reduce task would be the primary replica for these chunks. This is done by setting the top k bits of the chunk identifiers to be the coordinate of the desired server. Without failures, the two replicas are stored on one-hop neighbors of the primary replica and a different network link is used for each one.

When a job request is received by a front-end server, it is broadcast to all the other servers using the intra-CamCube network. When a server receives the broadcast request it determines if, for any of the input chunks, it is the primary replica and if so initiates a map task to process the locally stored file. However, any server can run a map task on any file, even though it is not storing a replica of the file. This is required, for example, to handle stragglers. The intermediate data generated by the map task is not stored in the distributed store but is sorted and written to a local disk, as with MapReduce and Hadoop. Each output is tagged with a 160-bit identifier, *mapTaskId*, that is equal to the chunkId of the input chunk processed by that task. If multiple map tasks run on the same server then multiple intermediate data files will be produced, each with a different mapTaskId. Once all the intermediate data has been written to disk the map phase has completed, and the shuffle and reduce phase commences. Camdoop supports both a synchronous start of the shuffle phase using an explicit message from a controller (to perform cluster wide scheduling) but also allows servers to independently asynchronously initiate the shuffle phase.

4.1 On-path aggregation

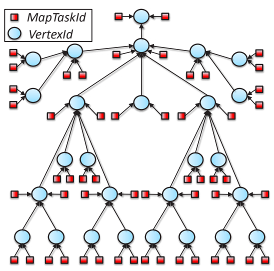
Camdoop pushes aggregation into the network and parallelizes the shuffle and reduce phases. To achieve this, it uses a custom transport service that provides reliable communication, application-specific scheduling of packets and packet aggregation across streams.

Conceptually, for each reduce task, the transport service forms a spanning tree that connects all servers, with the root being the server running the reduce task. This is similar to how overlay trees are used for performing convergecast [46]. This is hard to implement in traditional switched-based networks because servers have a single link to the switch; forwarding traffic through a server, even in the same rack, saturates the inbound link to the server [47]. It only makes sense when bandwidth oversubscription rates are very high.

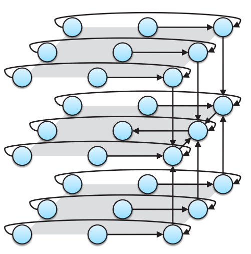
We start by describing a simplified version of the protocol used in Camdoop, assuming $R = 1$ and *no* failures, and then remove these assumptions.

Tree-building protocol Camdoop uses a tree topology similar to the one in Figure 3(a). In principle, any tree topology could be used but, for reasons that we will explain later, this (perhaps surprising) topology maximizes network throughput and load distribution. The internal vertices are associated with a 160-bit identifier, called *vertexId*. The leaves represent the outputs of the map tasks and are associated to the corresponding mapTaskId.

Each job uses N vertexIds. To ensure that each vertexId



(a) Logical view.



(b) Physical view.

Figure 3: Logical and physical view of the tree topology used in Camdoop on a 3x3x3 CamCube.

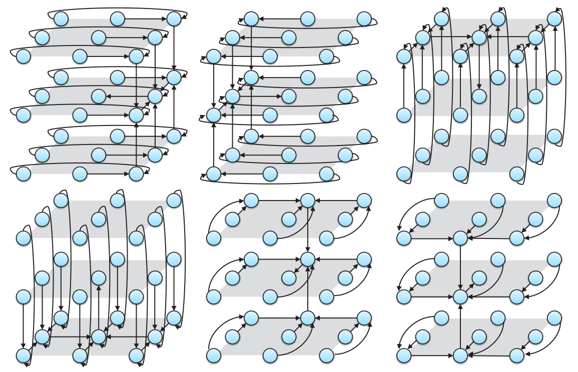


Figure 4: The six disjoint trees.

is mapped to a different server, a different coordinate is used for the top *k* bits of each vertexId. For the remaining bits, a hash of the jobId is used. The vertexId of the server selected to run the reduce task is denoted as *rootId*.

Servers use the function *getParent* to compute the tree topology. It takes as input the rootId and an identifier *i* (either a mapTaskId or a vertexId) and returns the identifier of the parent of *i*, or a null value if $i = rootId$. To achieve high throughput, the implementation of this function must ensure high locality. When *getParent* receives as input a mapTaskId, it looks at the coordinate generated from its top *k* bits and returns the vertexId that generates the same coordinate. Instead, when a vertexId *v* is passed to *getParent*, it always returns a vertexId *p* such that the coordinate of *v* is a one-hop neighbor of the coordinate of *p* in the 3D space. These conditions ensure that, in the absence of failures, map outputs are always read from the local disk (rather than from the network) and parents and children of the internal vertices are one-hop neighbors.

Figure 3(b) illustrates how the logical topology in Figure 3(a) is mapped onto a 3x3x3 CamCube using the mechanism just described (mapTaskIds omitted for clarity). This can be trivially extended to handle larger scales.

Shuffle and reduce phase When a server receives the job specification, it locally computes the list of the vertexIds of the job and identifies the subset that are mapped to itself. In the absence of failures, exactly one vertexId is mapped to a server. Then, by using the above function it computes the identifiers of its parent and children.

During the shuffle phase, the leaves of the tree (i.e., the mapTaskIds) just greedily send the sorted intermediate data to their parent identifiers, using the CamCube key-based routing. Every internal vertex merges and aggregates the data received by its children. To perform this efficiently, per child, a small packet buffer is maintained with a pointer to the next value in the packet to be aggregated. When at least one packet is buffered from each child, the server starts aggregating the (key,value) pairs across the packets using the combiner function. The aggregate (key,value) pairs are then sent to the parent. At the root, the results are aggregated using the reduce function and the results stored.

If a child and a parent identifier are mapped to the same physical server, a loopback fast-path is used. Otherwise, the transport service used in Camdoop provides a reliable in-order delivery of packets at the parent. Camdoop uses a window-based flow-control protocol and window update packets are sent between a parent and a child.

Load balancing and bandwidth utilization The described approach does not evenly distribute load across the tree, as some vertices have higher in-degree than others. Further, it is not able to exploit all six outbound links, as each server has only one parent to which it sends packets over a single link. To address these issues, Camdoop creates six independent *disjoint* spanning trees, all sharing the same root. The trees are constructed such that every vertexId, except the root, has a parent vertexId on a distinct one-hop server for each of the six trees. Conceptually, this can be achieved by taking the topology in Figure 3(b) and rotating it along the *Y* and *Z* axis. The resulting trees are shown in Figure 4. This explains the choice of the topology used in Figure 3(a) as it enables the use of six disjoint trees.

This ensures that, except for the outbound links of the root (which are not used at all), each physical link is used by exactly one *(parent, child)* pair in each direction. This enables the on-path aggregation to potentially exploit the 6 Gbps of inbound and outbound bandwidth per server. It also improves load balancing; the resources contributed are more uniformly distributed across servers. The state shared across trees is minimal and they perform the aggregation in parallel, which further improves performance.

When using multiple trees, the intermediate data stored at each server is striped across the trees. This requires that *i)* the keys remain ordered within each stripe and that *ii)* the mapping between key and stripe be performed consistently across all servers, so that the same keys are always forwarded through the same stripe. Camdoop applies a hash function to the keys so as to roughly distribute them uniformly across the six stripes. In general, order-preserving hash functions cannot be used, because the key distribution might be skewed. Therefore, at the root, the six streams and the local map output need to be merged together to provide the final output. If the key distribution is known in advance, a more efficient solution is to split the keyspace in six partitions of equal size such that the order of the keys is preserved across partitions and assign each partition to a different tree. In this way, the root just needs to merge the stream of each tree with the local map output data (which would have also been partitioned accordingly) and then concatenate the resulting streams, without requiring a global merge. However, since this approach is not always applicable, in all the experiments presented in the next section, the hash-based approach is used.

A drawback of using six disjoint trees is that this increases the packet hop count compared to using a single shortest-path tree. As we will see in Section 5.2, when R is very large and there is little opportunity for aggregating packets, this negatively impacts the performance. In the design of Camdoop, we preferred to optimize towards scenarios characterized by high aggregation and/or low number of reduce tasks and this motivates our choice of using six trees. However, if needed, a different tree topology could be employed to obtain different tradeoffs.

Multiple reduce tasks and multiple jobs Handling $R > 1$ is straightforward: each reduce task is run independently, and hence six disjoint trees are created *per reduce task*. This means that, in the absence of failures, each link is shared by R trees, i.e., one per reduce task. Each tree uses a different packet queue and the CamCube queuing mechanism ensures fair sharing of the links across trees. The same mechanism is also used to handle multiple jobs. Each job uses a different set of queues and the bandwidth is fair-shared across jobs.

Packet queue sizes are controlled by an adaptive protocol that evenly partitions the buffer space between the multiple reduce tasks and jobs. This ensures that the aggregate memory used by Camdoop to store packets is constant, regardless the number of tasks running. In the experiments presented in Section 5, the aggregate memory used by the packet queues was 440 MB per server.

4.2 Incorporating fault-tolerance

A key challenge in the design of on-path aggregation is to make it failure tolerant, and in particular to ensure that during failures we do not double count (key,value) pairs. We first describe how we handle link failures and then we discuss how we deal with server failures.

Handling link failures is easy. To route packets from children to parents, we use the CamCube key-based routing service, which uses a simple link-state routing protocol and shortest-paths. In case of link failures, it recomputes the shortest path and reroutes packets along the new path. While this introduces a path stretch (parents and children may not be one-hop neighbors any longer), due to the redundancy of paths offered by the CamCube topology, this has low impact on performance, as we show in Section 5.4. Our reliable transmission layer recovers any packets lost during the routing reconfiguration.

We now focus on server failures. The CamCube API guarantees that, in case of server failures, vertices are remapped to other servers in close proximity to the failed server. The API also notifies all servers that a server has crashed and that its vertex has been remapped. When the parent of the vertex v that was mapped to the failed server receives the notification, it sends a control packet to the server that has now become responsible for v. This packet contains the last key received from the failed server. Next, each child of v is instructed to re-send all the (key,value) pairs from the specified last key onwards. Since keys are ordered, this ensures that the aggregation function can proceed correctly. If the root of the tree fails, then all (key,value) pairs need to be re-sent, and the new vertex simply requests each child to resend from the start.

Of course, as in all implementations of MapReduce where the intermediate data is not replicated, if the failed server stored intermediate data, it will need to be regenerated. Some of the children of the failed vertex represent the map tasks that originally ran on the failed server, each identified by a different mapTaskId. On a failure, the CamCube API ensures that these identifiers are remapped to active servers. When these servers receive the control packet containing the last key processed by the parent of the failed vertex, they start a new map task using the mapTaskId to select the correct input chunk (recall that the chunkId is equal to the mapTaskId). If the distributed file system is used, the server to which the mapTaskId is remapped is also a secondary replica for that chunk. This ensures that, even in case of failure, the map tasks can read data locally. As the map function is deterministic, the sequence of keys generated would be the same as the one generated by the failed server, so the map task does not re-send (key,value) pairs before the last key known to have been incorporated.

4.3 Non commutative / associative functions

Although Camdoop has been designed to exploit on-path aggregation, it is also beneficial when aggregation cannot be used, i.e., when the reduce function is not commutative and associative (e.g., computing the median of a set of values). In these cases, Camdoop vertices only merge the streams received from their children, without performing any partial aggregation. Although this does not reduce the total amount of data routed, it distributes the sort load. In traditional MapReduce implementations, each reduce task receives and processes $N-1$ streams of data plus the locally stored data. In Camdoop, instead, each reduce task only merges 6 streams, and the local data. Even when $R < N$ all servers participate in the merge process.

Also, in Camdoop the computation of the reduction phase has been parallelized with the shuffle phase. Since all the streams are ordered by key, as soon as the root receives at least one packet from each of its six children, it can immediately start the reduce task without waiting for all packets. This also implies that there is no need to write to disk the intermediate data received by the reduce task, which further helps performance. Beside reducing overall job time, maximizing concurrency between the shuffle and reduce phase helps pipelining performance where the final output is generated as the result of a sequence of MapReduce jobs. As the reduce tasks produce pairs, the next map function can be immediately applied to the generated pair.

5 Evaluation

We evaluate the performance of Camdoop using a prototype 27-server CamCube and a packet-level simulator to demonstrate scaling properties.

Testbed The 27 servers form a 3x3x3 direct-connect network. Each server is a Dell Precision T3500 with a quad-core Intel Xeon 5520 2.27 GHz processor and 12 GB RAM, running an unmodified version of Windows Server 2008 R2. Each server has one 1 Gbps Intel PRO/1000 PT Quadport NIC and two 1 Gbps Intel PRO/1000 PT Dual-port NICs, in PCIe slots. We are upgrading the platform to use 6-port Silicom PE2G6i cards, but currently only have them in sample quantities, so in all our experiments we used the Intel cards. One port of the four port card is connected to a dedicated 48-port 1 Gbps NetGear GS748Tv3 switch (which uses store-and-forward as opposed to cut-through routing). Six of the remaining ports, two per multiport NIC, are used for the direct-connect network.

The Intel NICs support jumbo Ethernet frames of 9,014 bytes (including the 14 byte Ethernet header). In CamCube experiments we use jumbo frames and use default settings for all other parameters on the Ethernet cards, including interrupt moderation. We analyzed the performance of the switch using jumbo frames and found that the performance was significantly worse than using the traditional 1,514 byte Ethernet frames so all switch-based experiments use this size.

Each of the servers was equipped with a single standard SATA disk, and during early experiments we found that the I/O throughput of the disk subsystem was the performance bottleneck. We therefore equipped each server with an Intel X25-E 32 GB Solid State Drive (SSD). These drives achieve significantly higher throughput than the mechanical disks. We used these disks to store all input, intermediate and output data used in the experiments.

Simulator Our codebase can be compiled to run either on the CamCube runtime or on a packet-level discrete event simulator. The simulator accurately models link properties, using 1 Gbps links and jumbo frames. The simulator assumes no computation overhead. We ran simulations with 512 servers, representing an 8x8x8 CamCube. This is the same order of magnitude as the average number of servers used per MapReduce job at Google, which are 157, 268 and 394 respectively for the three samples reported in [21]. Also, anecdotally, the majority of organizations using Hadoop use clusters smaller than a few hundred servers [5].

Baselines Compared to existing solutions, Camdoop differs in two ways. First, it uses the CamCube direct-connect topology, running an application specific routing and transport protocol. Second, it exploits on-path aggregation to distribute the aggregation load and to reduce the traffic. To quantify these benefits separately, we implemented two variants of Camdoop, *TCP Camdoop* and *Camdoop (no agg.)*, which we use as baselines. Both variants run the same code as Camdoop for the map and reduce phase, including the ability to overlap the shuffle and reduce phase but they do not aggregate packets on path. The difference between the two baselines lies in the network infrastructure and protocols used. TCP Camdoop transfers packets over the switch using TCP/IP. The Camdoop (no agg.) baseline, instead, runs on top of CamCube and it uses the same tree-based routing and transport protocol used by Camdoop. It also partially sorts the streams of data on-path but it does not aggregate them.

By comparing the performance of Camdoop (no agg.) against TCP Camdoop, we can quantify the benefit of running over CamCube and using custom network protocols. This also shows the benefits of Camdoop when on-path aggregation cannot be used as discussed in Section 4.3. The comparison between Camdoop and Camdoop (no agg.) shows the impact of on-path aggregation.

In our experiments, across a wide range of workloads and configuration parameters, Camdoop significantly outperformed the other two implementations. To demonstrate that the performance of the switch-based implementation is good, we also compared its performance against two production systems: Apache Hadoop [1] and Dryad/DryadLINQ [30, 48]. As we show next, all Camdoop versions, including the one running over the switch, outperform them.

5.1 Sort and Wordcount

We evaluate all the three versions of Camdoop against Hadoop and Dryad/DryadLINQ running over the switch. We use two different jobs: `Sort` and `Wordcount`, chosen as they are standard tutorial examples included in the Hadoop and DryadLINQ distributions and they are often used to benchmark different MapReduce implementations. For Sort, we used the 'Indy' variant of the sort benchmark [6] in which the input data consists of randomly distributed records, comprising a 10 byte key and 90 bytes of data. Each key is unique and the aim of the Sort is to generate a set of output files such that concatenating the files generates a total ordering of the records based on the key values. In Wordcount the aim is to count the frequency with which words appear in a data set

These represent very different workloads. In Sort the input, intermediate and output data sizes are the same: there

	Sort		Wordcount	
	R=1	*R*=27	*R*=1	*R*=27
Hadoop	242.23	34.67	311.63	199.61
Dryad	n/a	16.14	n/a	10.57
TCP Camdoop	49.24	6.68	3.82	0.68
Camdoop (no agg.)	13.59	1.42	2.79	0.34
Camdoop	14.08	1.54	1.67	0.21

Table 1: Sort and Wordcount shuffle and reduce time (s).

is no aggregation of data across phases. In contrast, in Wordcount there is significant aggregation due to multiple occurrences of the same word in the original document corpus and in the intermediate data. From the statistics reported in [18, 21], we expect most workloads to have aggregation statistics closer to Wordcount.

We used Hadoop version 0.20.2 with default settings. We configured it so that all 27 servers could be used for map and reduce tasks, with one server running both the HDFS master and MapReduce job tracker. We increased the block size of HDFS to 128 MB because this yielded higher performance.

We used Dryad/DryadLINQ with default settings. DryadLINQ generates a Dryad dataflow graph which controls the number of instances of each type of process; therefore we could not vary this configuration parameters for the Dryad results. Further, in Dryad the master node cannot be used to run tasks. We therefore selected one server as the master and used the other 26 servers as workers. The impact of this is that there were fewer resources to be used to process the jobs. Therefore, to be conservative we automatically scaled the input data sets to be 26/27ths of the data set used for Camdoop and Hadoop. This effectively reduces the amount of work needed to be performed by Dryad by the load sustained on one server in the other implementations. In order to facilitate comparisons, in all Camdoop-based implementations we write the results of the reduce task directly to the local disk and we use no replication in Hadoop and Dryad/DryadLINQ so that results are also written only to the local disk. In all experiments, we measure the *job time*, which we define as the time from when the job is submitted to when the final results are stored on disk, and the *shuffle and reduce time*, which is the time from when all map tasks have completed till the final results are stored on disk. The results in Table 1 show the shuffle and reduce time for all the implementations using *R*=1 and *R*=27 reduce tasks. For Dryad, we show only one result using the default data flow graph produced by DryadLINQ.

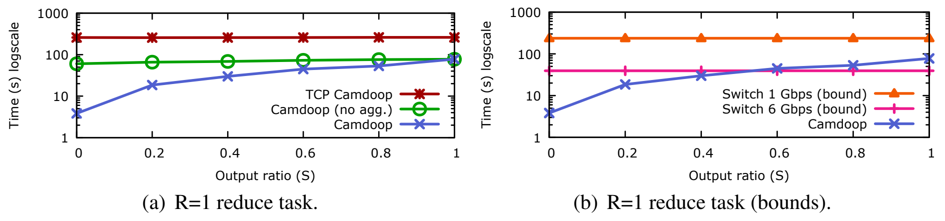
The Sort input data consists of 56,623,104 records. For Hadoop, these are randomly distributed in files stored in HDFS based on the block size. For Dryad and Camdoop we created files of approximately 200 MB and distributed them across the 27 servers (resp. 26 servers for Dryad). In this job, Camdoop and Camdoop (no agg.) demonstrate similar performance as there is no possibility to perform aggregation. For TCP Camdoop the 1 Gbps link to the switch for the server running the reduce task is the bottleneck and limits performance. When running with *R*=27, the TCP Camdoop version does not fully exploit the server bandwidth due to the overhead of managing multiple concurrent TCP flows. We will demonstrate this in more detail later. The versions running on CamCube are able to exploit the higher bandwidth available, and therefore perform better than TCP Camdoop but are constrained by the SSD disk bandwidth.

The results in Table 1 show that all Camdoop versions, including TCP Camdoop, outperform Hadoop and Dryad. Part of the performance gains is due to the design choices of Camdoop, most prominently the ability of overlapping the shuffle and the reduce phase (Section 4.3), which also reduces disk IO. We also finely optimized our implementations to further improve the performance. In particular, Camdoop mostly utilizes unmanaged memory and statically allocated packet buffers to reduce the pressure on the garbage collector, it exploits the support of C# for pointer operations to compare MapReduce keys eight bytes at a time rather than byte-by-byte, and, finally, it leverages an efficient, multi-threaded, binary merge-sort to aggregate streams. However, we also note that our versions are prototype implementations, omitting some functionality found in a full production system. This can also partly explain the difference in the results. Finally, we made no effort to tune Hadoop and Dryad but we used the default configurations. We show the data point simply to validate that the performance of TCP Camdoop is reasonable and we consider it as our main baseline.

For the Wordcount job the input data is the complete dump of the English language Wikipedia pages from 30th January 2010, and consists of 22.7 GB of uncompressed data. For Dryad and Camdoop implementations we split the file into 859 MB files and distributed one to each of the 27 servers (resp. 26 for Dryad). For Hadoop we stored these in HDFS using the modified block size. We also use map-side combiners in both Hadoop and Dryad. In this case, the intermediate data for the Wordcount can be aggregated. As in Sort, in Table 1 we show the shuffle and reduce time for all implementations, including the data points for Hadoop and Dryad to validate that the performance of TCP Camdoop is reasonable. In the Wordcount job the interesting comparison is between Camdoop and Camdoop (no agg.). Regardless the value of *R*, Camdoop is using *all* the servers to help perform the aggregation, which results in less data and reduce overhead at the server running the reduce task. This yields a factor of 1.67 (*R*=1) and 1.62 (*R*=27) reduction in shuffle and reduce time.

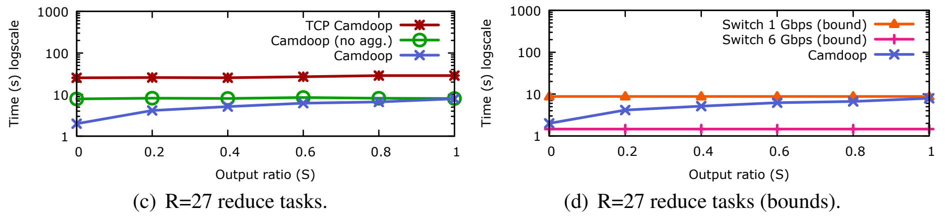
5.2 Impact of aggregation

In order to conduct experiments across varying configuration parameters we created a synthetic job, inspired by the Wordcount example. We chose this type of job be-



(a) R=1 reduce task.

(b) R=1 reduce task (bounds).



(c) R=27 reduce tasks.

(d) R=27 reduce tasks (bounds).

Figure 5: Varying the output ratio using R=1 and R=27 reduce tasks on the 27-server testbed (log-scale).

cause, despite its simplicity, it is representative of many, more complex, jobs in which aggregation functions typically consist of simple additive or multiplicative integer or floating point operations, e.g., data mining or recommendation engines. The input data consists of 22.2 GB of data partitioned into 27 files, each 843 MB in size. The data consists of strings, generated randomly with a length uniformly selected at random from 4 and 28 characters. The job uses a map function that takes each key and generates a key value pair of (key, 1). There is no compression of input data to intermediate data, and the intermediate data will add a four-byte counter to each key. The reduction function sums the set of counts associated with a key, and can also be used as the combiner function. For experiments on the simulator, to allow us to scale, we use a smaller data set file size of 2 MB per server, which in our 512-server setup yields a total input data size of 1 GB.

To explore the parameter space we generate input data set files by specifying an output ratio, S. We define S as the ratio of the output data set size to the intermediate data set size. If there are N input files the lower bound on S is $1/N$ and the maximum value is S=1. For clarity, we adopt the notation that S=0 means $S = 1/N$. When S=1 there is no similarity between the input files, meaning that each key across all data files is unique, and there is no opportunity to perform on-path aggregation. Subsequently, at the end of the reduction phase the size of the output files will be the union of all keys in the input files, with each key assigned the value 1. For 27 servers this will result in an output data set size of approximately 28.3 GB (including the four-byte counters for each key). This represents the *worst-case* for Camdoop because no aggregation can be performed. When S=0 each key is common to all input files, meaning that the output file will be approximately 1.1 GB with each key having the value 27. This represents workloads where we can obtain maximum benefit from on path aggregation. As a point of comparison the Sort represented a workload where S=1, and the Wordcount represented a workload with $S = 0.56$. A top-k query as used in search engines represents an example where S=0 because all map tasks generate a sorted list of k pairs and the output result is also a list of k pairs. By varying S, we are able to model different workloads, which result into different traffic savings and, hence, different performance gains.

In the results we do not consider the time taken by map tasks. Map times are independent of the shuffle and reduce phase. They can vary significantly based on implementation, and factors like the source of the input data, e.g. reading from an unstructured text data file as opposed to reading from (semi-)structured sources, such as BigTable or using binary formats [22].

In the previous experiments the output data was stored to SSDs. Profiling Camdoop showed that for some data points the performance of reduce tasks was impacted by the SSD throughput by as much as a factor of two. We consider this a function of provisioning of the servers, and it would be very feasible to add a second SSD to increase the throughput so it would not impact performance. However, as we were unable to do this, in these experiments we do not write the final output to disk. We do read all intermediate data from disk.

Impact of output ratio First we examine the impact of varying the output ratio S from S=0 to 1 on the testbed and on the simulator. When S=0, full aggregation is possible, and when S=1 no aggregation is possible. We use two values of R, R=1 and R=N, i.e., the lower and upper bounds on R assuming we have at most one reduce task per server.

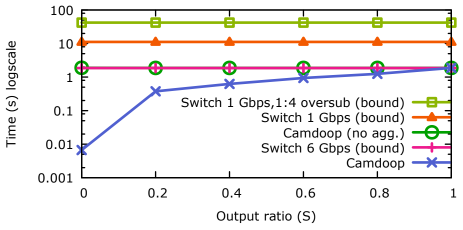
Figure 5 shows the shuffle and reduce time on the testbed as we vary S, using a log-scale y-axis. We compare Camdoop against Camdoop (no agg.) and TCP Camdoop. Figure 5(a) shows the results for R=1 and Figure 5(c) shows the results for R=27. In running the experiments for TCP Camdoop, we observed that the TCP throughput dropped for large values of R. We were able to replicate the same behavior when generating all-to-all traffic using `ttcp`. Hence, this is not an artifact of our

TCP Camdoop implementation. We speculate that this is due to a combination of the small switch buffers [13] and the performance issues of TCP when many flows share the same bottleneck [34]. Potentially, this overhead could be partly reduced with a fine-tuned implementation and using a high-end switch. For completeness, in Figure 5(b) and 5(d) we report the *lower bound* of the time that would be required by *any* switch-based implementation. This is computed by dividing the data size to be shuffled by the server link rate (1 Gbps). This therefore assumes full-bisection bandwidth and it does not include reduce time. As a further point of comparison, we also include the lower bound for a switch-based configuration with *six* 1 Gbps links teamed together per server. Although the number of links per server is identical to CamCube, this configuration offers much higher bandwidth because in CamCube the server links are also used to forward traffic. Also, it would increase costs as it requires more switches and with higher fanout. We refer to the two lower bounds as *Switch (1 Gbps)* and *Switch (6 Gbps)*.

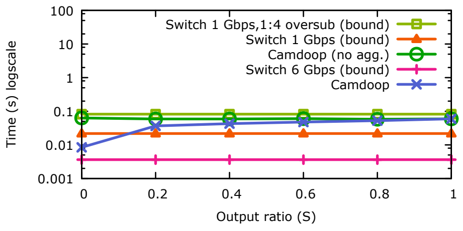
The first important result in Figure 5(a) and 5(c) is the difference between TCP Camdoop and Camdoop (no agg.). Camdoop (no agg.) achieves significantly higher performance across all values of S, both for R=1 and R=27. This is significant because it shows the base performance gain Camdoop achieves by using CamCube and a custom transport and routing protocol. In general, for Camdoop (no agg.) and TCP Camdoop the time taken is independent of S since the size of data transferred is constant across all values of S.

We now turn our attention to the comparison of Camdoop and Camdoop (no agg.). As expected, when S=1 the performance of Camdoop and Camdoop (no agg.) is the same, because there is no opportunity to aggregate packets on-path. As S decreases, the benefit of aggregation increases, reducing the number of packets forwarded, increasing the available bandwidth and reducing load at each reduce task. This is clearly seen in Figures 5(a) and Figure 5(c) for Camdoop for R=1 and R=27. When S=0, Camdoop achieves a speedup of 12.67 for R=1 and 3.93 for R=27 over Camdoop (no agg.) (resp. 67.5 and 15.71 over TCP Camdoop).

Figure 5(b) and 5(d) show that, at this scale, Camdoop *always* achieves a lower shuffle and reduce time than Switch (1 Gbps). When R=1, for low values of S Camdoop also outperforms Switch (6 Gbps), but for large values of S, Camdoop performance is bottlenecked on the rate at which the reduce task is able to process incoming data. In our implementation the throughput of the reduce code path varies approximately from 2.41 Gbps when S=0 to 3.15 Gbps when S=1). In a real implementation, the performance of Switch (6 Gbps) would suffer from the same constraint. When R=27, the shuffle and reduce time of Camdoop is always higher than Switch (6 Gbps) due to the lower bandwidth available in CamCube.



(a) R=1 reduce task.



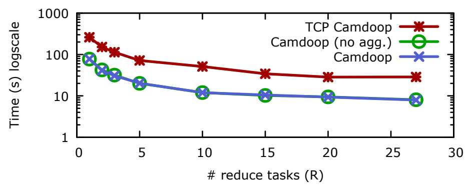
(b) R=512 reduce tasks.

Figure 6: Varying the output ratio using R=1 and R=512 reduce tasks in the 512-server simulation (log-scale).

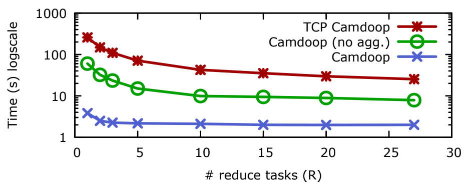
Figure 6 shows the simulation results. Due to the complexity of accurately simulating TCP at large scale, we only plot the values for the switch lower bounds. Building a full-bisection cluster at scale is expensive and most deployed clusters exhibit some degree of oversubscription, ranging from 1:4 up to 1:100 and higher. To account for this, we also compute the lower bound for a 512-server cluster assuming 40 servers per rack and 1:4 oversubscription between racks: *Switch (1 Gbps, 1:4 oversub)*.

The simulated results confirm what we observed on the testbed. When R=1, Camdoop *always* achieves the lowest shuffle and reduce time across *all* values of S. The simulation does not model computation time, which explains why Camdoop outperforms Switch (6 Gbps) even for higher values of S. When R=512, Switch (6 Gbps) always yields the lowest time due to the higher bisection bandwidth. More interesting is the comparison between Camdoop and Switch (1 Gbps) when R=512. Routing packets using six disjoint trees increases the path hop count, thus consuming more bandwidth. This explains why when R=512, for most values of S, Switch (1 Gbps) performs better than Camdoop. Using a single shortest-path tree would enable Camdoop, in this scenario, to achieve higher performance than Switch (1 Gbps) across all values of S. As discussed in Section 4.1, we decided to optimize Camdoop for scenarios with low R or low S. However, even with this suboptimal configuration, Camdoop is always better than Switch (1 Gbps, 1:4 oversub).

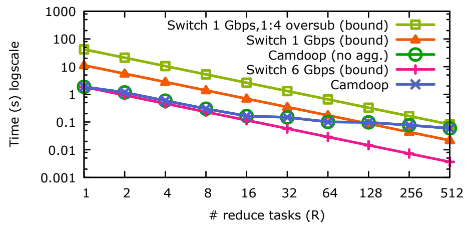
Impact of the number of reduce tasks In the previous experiment we discussed the impact of varying S when R=1 and $R = N$. In Figure 7, instead, we show the shuffle and reduce time on the testbed and on the simulator as we vary the number of reduce tasks, with and without aggregation. In these experiments, we used two data sets with



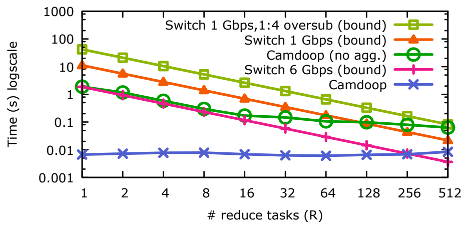
(a) S=1 (27-server testbed).



(b) S=0 (27-server testbed).



(c) S=1 (512-server simulation).



(d) S=0 (512-server simulation).

Figure 7: Varying the number of reduce tasks using S=0 and S=1 workloads (log-scale).

S=0 and *S*=1, representing the two ends of the spectrum for potential aggregation.

As already shown in the previous experiments, in Figure 7(a) when *S*=1 the performance of Camdoop and Camdoop (no agg.) is the same. They both benefit from higher values of *R*, as expected since the size of data received and processed by each reduce task decreases. They also outperform TCP Camdoop across all values of *R* due to the higher bandwidth provided by CamCube and the custom network protocols used. Similar trends are also visible in the simulation results for *S*=1 in Figure 7(c). Camdoop and Camdoop (no agg.) always outperform Switch (1 Gbps, 1:4 oversub) for all values of *R*. They also exhibit higher performance than Switch (1 Gbps) for most values of *R*, although for the reasons discussed above, with high values of *R* and *S*, using six trees is less beneficial.

Figure 7(b) shows the results for *S*=0. We already observed that the performance of Camdoop (no agg.) and TCP Camdoop is generally unaffected by the value of *S* while Camdoop significantly improves performance when *S*=0. However, the important observation here is that in this configuration the shuffle and reduce time for Camdoop is largely independent of *R* as shown in both Figure 7(b) and 7(d). This is because, regardless the value of *R*, *all* servers participate in aggregation and the load is evenly distributed. This can be observed by looking at the distribution of bytes aggregated by each server. For instance, when *R*=1, the minimum and maximum values of the distribution are within the 1% of the median value. This means that the shuffle and reduce time becomes a function of the output data size rather than the value of *R*. *R* just specifies the number of servers that store the final output. This is significant because it allows us to generate only *few* output files (possibly just one) while still utilizing *all* resources in the cluster. This property is important for multi-stage jobs and real time user queries.

The reason why in Figure 7(b) the shuffle and reduce time of Camdoop in the testbed with *R*=1 is higher than for *R* >1 is because the performance is bottlenecked by the computation rate of the reduce task. When *R* is higher, the size of the data aggregated by each reduce task is lower and, hence, the reduce task is not the bottleneck anymore. This effect is not visible in the simulation results because the simulator does not model computation time.

Finally, we note that in the simulation results in Figure 7(d), Camdoop outperforms even Switch (6 Gbps) for most values of *R*. This demonstrates that although CamCube provides less bandwidth than Switch (6 Gbps), by leveraging on-path aggregation, Camdoop is able to decrease the network traffic and, hence, to better use the bandwidth available and reduce the shuffle time.

The results show the benefits of using CamCube instead of a switch and also the benefits of on-path aggregation. We have so far shown results for experiments where only one job was running. However, often users run multiple jobs concurrently. In the next set of experiments we evaluate the performance of Camdoop with such workloads.

5.3 Partitioning

The next experiment evaluates the impact of running multiple jobs. We examine how this scales when using two different strategies for running multiple jobs, *horizontal* and *vertical* partitioning. In horizontal partitioning, we run multiple jobs by co-locating them. Hence, all servers run an instance of the map task for each job. In vertical partitioning, we divide the set of servers into disjoint partitions and run only the map tasks of any job on the servers assigned to it. In the horizontal partitioning, we run between 1 and *N* reduce tasks, but the tasks are scheduled such to minimize the load skew across all servers. In vertical partitioning reduce tasks are only scheduled on the servers assigned to that job. Note that in vertical partitioning servers in each partition contribute resources to both

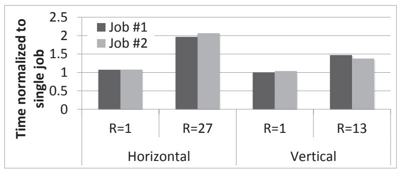


Figure 8: Effect of running multiple jobs.

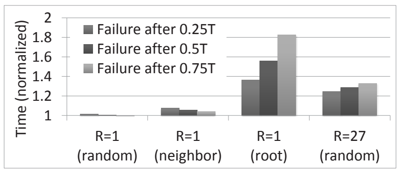


Figure 9: Increase in shuffle and reduce time caused by server failure.

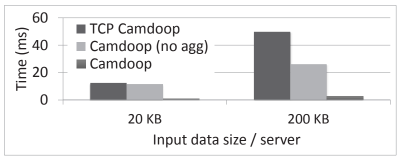


Figure 10: Shuffle and reduce time for small input data sizes.

jobs, as *all* servers forward and aggregate packets on-path.

Our experiments run two identical jobs concurrently. We start both jobs at the same time and we use *S*=1, which is the worst case. For horizontal partitioning we set *R*=1 and *R*=27, and have all servers run instances of the map task. In vertical partitioning we randomly assign thirteen servers to each job, leaving one server unassigned.

Figure 8 shows the *normalized* shuffle and reduce time for multiple jobs using horizontal and vertical partitioning. For horizontal partitioning the values are normalized with respect to running a single job on all 27 servers with *R*=1 or *R*=27, as appropriate. For vertical partitioning the values are normalized with respect to running a single job on 13 servers with *R*=1 or $R = 13$, as appropriate.

The first important observation is that in all configurations, the time taken by the two jobs is approximately the same. This demonstrates the ability of CamCube to fair-share the bandwidth between two jobs by using separate outbound packet queues, as described in Section 4.1.

The results for horizontal partitioning in Figure 8 show that when *R*=1 the time taken for *two* jobs is approximately at most 1.07 times longer than to complete a *single* job. When *R*=1 the bottleneck is the rate at which the reduce task is able to process incoming data. In our testbed, the reduce code path can sustain a throughput of approximately 3.15 Gbps. The available bandwidth is 6 Gbps, and hence, in general, the CamCube links are under-utilized. The links into the server running the reduce task have the highest utilization rate at 50.8%. Running multiple jobs allows the extra capacity to be utilized, and explains why the time to run two jobs is only less than 1.1 times longer than running a single job. In contrast, when *R*=27, all links are fully utilized. Adding the second job causes the time to double because the links need be shared between jobs.

When using vertical partitioning and *R*=1, the two jobs take approximately as long as running a single job using 13 servers, as expected because the single experiment uses 50% of the CamCube resources. When *R*=13, intuitively you would expect the same, with the two jobs executing in the same time as the single task. However, in this case where *R*=13 there is higher link utilization, and adding a second job increases job time up to a factor of 1.47.

The non-normalized figures show that the maximum time for two jobs when $R = 13$ is comparable to one job with *R*=27, as would be expected. This demonstrates that there is no performance difference between running jobs horizontally or vertically partitioned. The achieved results are simply a function of the intermediate data set size and number of reduce tasks. This is important, because we envisage in many scenarios where the input data is stored in a key-value store that distributes input records across all servers in the CamCube cluster. Hence, running with horizontal partitioning will likely be the norm.

5.4 Failures

We now consider the impact of server failures on the performance of Camdoop. Camdoop uses a tree mapped onto the CamCube which, without failures, ensures that an edge in the tree is a single one-hop link in the CamCube and each vertex is mapped to a different server. The failure of a server breaks this assumption. A single edge in the tree becomes a multi-hop path in the CamCube. Also, the load on other servers increases as they need to perform the aggregation on behalf of the failed server. This persists until the server is replaced, assuming it is replaced. Also, if the failure occurs during the shuffle phase, data that has been lost needs to be re-sent.

In this experiment, we want to measure the efficiency of the recovery protocol detailed in Section 4.2 and the increase in the job execution time due to a server failing during the shuffle phase. We test this on the testbed as the simulator does not model computational overhead. We fail a server after the shuffle phase has started and report the total shuffle and reduce time (including the time elapsed before the failure occurred) normalized by the time T taken in a run with no failures. To evaluate the impact of the time at which a server fails, we repeated the experiment by failing the server respectively after $0.25 \cdot T$ seconds, $0.5 \cdot T$ seconds, and $0.75 \cdot T$ seconds. We use *S*=1 as this is the worst case. When *R*=1 we consider the impact of *i)* failing the root of the tree (i.e., the server running the reduce task), *ii)* failing a neighbor of the root, and *iii)* a random server that is neither the root nor a neighbor of the root. When *R*=27, since all servers run reduce tasks, failing a random server will fail the root of one tree and a neighbor of six other roots.

As explained in Section 4.2, when a server fails, all the map tasks that it was responsible for need to be re-executed elsewhere because their outputs have been lost (recall that as in MapReduce the output of the map tasks is not replicated). The time taken to re-run the map tasks can mask the actual performance of the recovery protocol.

To avoid this effect, in these experiments we configured the job with only 26 map tasks and ensured that no map task is assigned to the failing server.

Figure 9 shows the shuffle and reduce time normalized against T. When R=1 the bottleneck is the rate at which the reduce task can aggregate data on the 6 inbound links. As with the multi-job scenario, under-utilized links mean the impact of failing a *random* server is minimal. The load of the failed server is distributed across multiple servers, as the vertexes that were running on the failed server are automatically mapped to different servers. Also, by leveraging the knowledge of the last key received by the parent of the failed vertex, only few extra packets need be re-sent.

Failing a *neighbor* server of the root has higher impact because the incoming bandwidth to the root is reduced, but it is still low as the reduction in bandwidth to the root server is 1/6th. The earlier the failure occurs, the lower bandwidth will be available in the rest of the phase. This explain why the time stretch is higher when a failure occur at $0.25 \cdot T$ rather than later.

When the *root* fails, the time significantly increases because all data transferred before the failure occurred have been lost and, hence, the shuffle phase needs to restart from the beginning. Unlike the neighbor failure experiment, the time stretch when a root fails is higher when the failure occurs towards the end of the shuffle phase rather than at the beginning. The reason is that the amount of work that has been wasted and must be repeated is linearly proportional to the time at which the failure occurs. This is confirmed by the results in Figure 9. For instance, failing the root after $0.25 \cdot T$ increases the shuffle and reduce time by a factor of 1.36 while if the root fails after $0.75 \cdot T$ the time increases by 1.83. For similar reasons when R=27 the stretch is proportional to the failure time, although its impact is limited because a server will be the root only for one tree and a neighbor for only 6 trees.

5.5 Impact of small input data size

In the last experiment, we want to evaluate the performance of Camdoop when the input data size is small. This is important for real time applications like search engines in which the responses of each server are of the order of a few tens to hundreds of kilobytes [13]. We ran an experiment similar to the ones in Section 5.2 but, instead of 843 MB, we used smaller file sizes of 20 KB and 200 KB respectively. We chose a workload with S=0 because, as already observed, typically the size of the output of each map task is equal to the size of the final results. Finally, we set R=1 as *required* by these applications.

Figure 10 shows the results for the three Camdoop implementations. For both input sizes, Camdoop outperforms Camdoop (no agg.) and TCP Camdoop, respectively by a factor of 13.42 and 14.33 (20 KB) and 9.05 and 17.23 (200 KB). These results demonstrate the feasibility of Camdoop even for applications characterized by small input data size and low latency requirements.

6 Related work

Aggregation has always been exploited in MapReduce, through the combiner function, to help reduce both network load and job execution time [21]. More recently, it was noted that, in bandwidth oversubscribed clusters, performance can be improved by performing partial aggregation at a rack-level [47]. However, at scale rack-level aggregation has a small impact on intermediate data size, so that high values of R are still required. Further, even at rack-level the 1 Gbps link for the aggregation server is a bottleneck. Camdoop builds on this work, and demonstrates that if you have a cluster architecture that enables in-network aggregation, e.g. the direct-connect topology of CamCube, it provides benefit at rack-scale and larger.

Recently, several extensions and optimizations to the MapReduce model have been proposed, including support for iterative jobs [35], incremental computations [16], and pipelining of the map and reduce phase [19]. These are not currently supported by our Camdoop prototype. However, these extensions are complementary to the ideas presented here and, like the standard MapReduce model, they would also benefit from in-network aggregation and could be integrated in Camdoop.

There have also been several proposals to improve the network topologies in data centers, including switch-based topologies [11, 26, 28, 40] and direct-connect (or hybrid) topologies [9, 27, 39]. Camdoop's design explicitly targets the CamCube topology and its key-based API. In principle, however, our approach could also be applied to other topologies in which servers can directly control routing and packet processing. This naturally includes direct-connect topologies, e.g., [39]. If traditional, hardware-based, routers were to be replaced by software routers [24, 32] or NetFPGAs [36], it could be possible to perform in-network aggregation also in switch-based topologies. This would require a considerable protocol re-engineering though, due to the higher node in-degree.

In-network aggregation has been successfully used in other fields, including sensor networks [29, 33], publish/subscribe systems [23, 43], distributed stream processing systems [8], and overlay networks [42, 46]. Inspired by these approaches, Camdoop leverages the ability of CamCube to process packets on path to reduce network traffic (and, hence, improve performance), without incurring the overhead and path stretch, typical of overlay-based approaches.

7 Conclusions

We have described Camdoop, a MapReduce-like system that exploits CamCube's unique properties to achieve high performance. We have shown, using a small prototype, that Camdoop running on CamCube outperforms Camdoop running over a traditional switch. We have also shown, using simulations, that these properties still hold at scale. Even if current clusters achieved full bisection bandwidth, Camdoop on CamCube would still outperform them in most scenarios, due to the ability of significantly reducing network traffic by aggregating packets on-path.

Acknowledgements We thank the reviewers, and in particular our shepherd, Emin Gün Sirer, who provided valuable feedback and advice.

References

[1] Apache Hadoop. `http://hadoop.apache.org/`.

[2] Big Data @ Foursquare . `http://goo.gl/FAmpz`.

[3] Big Data in Real Time at LinkedIn. `http://goo.gl/6OzCN`.

[4] Google Tree Distribution of Requests . `http://goo.gl/RpB45`.

[5] Hadoop Wiki: PoweredBy. `http://goo.gl/Bbfu`.

[6] Sort Benchmark Homepage. `http://sortbenchmark.org/`.

[7] Twitter Storm. `http://goo.gl/Y1AcL`.

[8] ABADI, D. J., AHMAD, Y., BALAZINSKA, M., CHERNIACK, M., HYON HWANG, J., LINDNER, W., MASKEY, A. S., RASIN, E., RYVKINA, E., TATBUL, N., XING, Y., AND ZDONIK, S. The Design of the Borealis Stream Processing Engine. In *CIDR* (2005).

[9] ABU-LIBDEH, H., COSTA, P., ROWSTRON, A., O'SHEA, G., AND DONNELLY, A. Symbiotic Routing in Future Data Centers. In *SIGCOMM* (2010).

[10] ADIGA, N. R., BLUMRICH, M. A., CHEN, D., COTEUS, P., GARA, A., GIAMPAPA, M. E., HEIDELBERGER, P., SINGH, S., STEINMACHER-BUROW, B. D., TAKKEN, T., TSAO, M., AND VRANAS, P. Blue Gene/L Torus Interconnection Network. *IBM Journal of Research and Development 49*, 2 (2005).

[11] AL-FARES, M., LOUKISSAS, A., AND VAHDAT, A. A Scalable, Commodity Data center Network Architecture. In *SIGCOMM* (2008).

[12] AL-FARES, M., RADHAKRISHNAN, S., RAGHAVAN, B., HUANG, N., AND VAHDAT, A. Hedera: Dynamic Flow Scheduling for Data Center Networks. In *NSDI* (2010).

[13] ALIZADEH, M., GREENBERG, A., MALTZ, D. A., PADHYE, J., PATEL, P., PRABHAKAR, B., SENGUPTA, S., AND SRIDHARAN, M. Data center TCP (DCTCP). In *SIGCOMM* (2010).

[14] BARROSO, L. A., AND HÖLZLE, U. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Morgan & Claypool Publishers, 2009.

[15] BENSON, T., AKELLA, A., AND MALTZ, D. A. Network Traffic Characteristics of Data Centers in the Wild. In *IMC* (2010).

[16] BHATOTIA, P., WIEDER, A., RODRIGUES, R., ACAR, U. A., AND PASQUIN, R. Incoop: MapReduce for Incremental Computations. In *SOCC* (2011).

[17] BORTHAKUR, D., GRAY, J., SARMA, J. S., MUTHUKKARUPPAN, K., SPIEGELBERG, N., KUANG, H., RANGANATHAN, K., MOLKOV, D., MENON, A., RASH, S., SCHMIDT, R., AND AIYER, A. Apache Hadoop Goes Realtime at Facebook. In *SIGMOD* (2011).

[18] CHEN, Y., GANAPATHI, A., R.GRIFFITH, AND KATZ, R. The Case for Evaluating MapReduce Performance Using Workload Suites. In *MASCOTS* (2011).

[19] CONDIE, T., CONWAY, N., ALVARO, P., HELLERSTEIN, J. M., ELMELEEGY, K., AND SEARS, R. MapReduce Online. In *NSDI* (2010).

[20] COSTA, P., DONNELLY, A., O'SHEA, G., AND ROWSTRON, A. CamCube: A Key-based Data Center. Tech. rep., MSR, 2010.

[21] DEAN, J., AND GHEMAWAT, S. MapReduce: Simplified Data Processing on Large Clusters. *Comm. of ACM 51*, 1 (2008).

[22] DEAN, J., AND GHEMAWAT, S. MapReduce: A Flexible Data Processing Tool. *Comm, of ACM 53*, 1 (2010).

[23] DEMERS, A., GEHRKE, J., HONG, M., PANDA, B., RIEDEWALD, M., SHARMA, V., AND WHITE, W. Cayuga: A General Purpose Event Monitoring System. In *CIDR* (2007).

[24] DOBRESCU, M., EGI, N., ARGYRAKI, K., CHUN, B.-G., FALL, K., IANNACCONE, G., KNIES, A., MANESH, M., AND RATNASAMY, S. RouteBricks: Exploiting Parallelism To Scale Software Routers. In *SOSP* (2009).

[25] GHEMAWAT, S., GOBIOFF, H., AND LEUNG, S.-T. The Google File System. In *SOSP* (2003).

[26] GREENBERG, A., HAMILTON, J. R., JAIN, N., KANDULA, S., KIM, C., LAHIRI, P., MALTZ, D. A., PATEL, P., AND SENGUPTA, S. VL2: A Scalable and Flexible Data Center Network. In *SIGCOMM* (2009).

[27] GUO, C., LU, G., LI, D., WU, H., ZHANG, X., SHI, Y., TIAN, C., ZHANG, Y., AND LU, S. BCube: A High Performance, Server-centric Network Architecture for Modular Data Centers. In *SIGCOMM* (2009).

[28] GYARMATI, L., AND TRINH, T. A. Scafida: A Scale-Free Network Inspired Data Center Architecture. *SIGCOMM Computer Communication Review 40* (2010).

[29] INTANAGONWIWAT, C., GOVINDAN, R., ESTRIN, D., HEIDEMANN, J., AND SILVA, F. Directed Diffusion for Wireless Sensor Networking. *IEEE/ACM TON 11*, 1 (2003).

[30] ISARD, M., BUDIU, M., YU, Y., BIRRELL, A., AND FETTERLY, D. Dryad: Distributed Data-parallel Programs from Sequential Building Blocks. In *EuroSys* (2007).

[31] KANDULA, S., PADHYE, J., AND BAHL, P. Flyways To De-Congest Data Center Networks. In *HotNets* (2009).

[32] LU, G., GUO, C., LI, Y., ZHOU, Z., YUAN, T., WU, H., XIONG, Y., GAO, R., AND ZHANG, Y. ServerSwitch: A Programmable and High Performance Platform for Data Center Networks. In *NSDI* (2011).

[33] MADDEN, S., FRANKLIN, M. J., HELLERSTEIN, J. M., AND HONG, W. TAG: a Tiny AGgregation Service for Ad-hoc Sensor Networks. In *OSDI* (2002).

[34] MORRIS, R. TCP Behavior with Many Flows. In *ICNP* (1997).

[35] MURRAY, D. G., SCHWARZKOPF, M., SMOWTON, C., SMITH, S., MADHAVAPEDDY, A., AND HAND, S. CIEL: A Universal Execution Engine for Distributed Data-Flow Computing. In *NSDI* (2011).

[36] NAOUS, J., GIBB, G., BOLOUKI, S., AND MCKEOWN, N. NetFPGA: Reusable Router Architecture for Experimental Research. In *PRESTO* (2008).

[37] PARHAMI, B. *Introduction to Parallel Processing: Algorithms and Architectures*. Kluwer Academic Publishers, 1999.

[38] RATNASAMY, S., FRANCIS, P., HANDLEY, M., KARP, R., AND SHENKER, S. A Scalable Content-addressable Network. In *SIGCOMM* (2001).

[39] SHIN, J.-Y., WONG, B., AND SIRER, E. G. Small-world Datacenters. In *ACM SOCC* (2011).

[40] SINGLA, A., HONG, C.-Y., POPA, L., AND GODFREY, P. B. Jellyfish: Networking Data Centers Randomly. In *NSDI* (2012).

[41] THUSOO, A., SHAO, Z., ANTHONY, S., BORTHAKUR, D., JAIN, N., SEN SARMA, J., MURTHY, R., AND LIU, H. Data Warehousing and Analytics Infrastructure at Facebook. In *SIGMOD* (2010).

[42] VAN RENESSE, R., BIRMAN, K. P., AND VOGELS, W. Astrolabe: A Robust and Scalable Technology for Distributed System Monitoring, Management, and Data Mining. *TOCS 21*, 2 (2003).

[43] VAN RENESSE, R., AND BOZDOG, A. Willow: DHT, Aggregation, and Publish/Subscribe in One Protocol. In *IPTPS* (2004).

[44] VASUDEVAN, V., PHANISHAYEE, A., SHAH, H., KREVAT, E., ANDERSEN, D. G., GANGER, G. R., GIBSON, G. A., AND MUELLER, B. Safe and Effective Fine-grained TCP Retransmissions for Datacenter Communication. In *SIGCOMM* (2009).

[45] WU, H., FENG, Z., GUO, C., AND ZHANG, Y. ICTCP: Incast Congestion Control for TCP in Data Center Networks. In *CoNEXT* (2010).

[46] YALAGANDULA, P., AND DAHLIN, M. A Scalable Distributed Information Management System. In *SIGCOMM* (Aug. 2004).

[47] YU, Y., GUNDA, P. K., AND ISARD, M. Distributed Aggregation for Data-Parallel Computing: Interfaces and Implementations. In *SOSP* (2009).

[48] YU, Y., ISARD, M., FETTERLY, D., BUDIU, M., LFAR ERLINGSSON, GUNDA, P. K., AND CURREY, J. DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language. In *OSDI* (2008).

WiFi-NC : WiFi Over Narrow Channels

Krishna Chintalapudi[⋆], Bozidar Radunovic[†], Vlad Balan[‡], Michael Buettener[§],
Srinivas Yerramalli[‡], Vishnu Navda[⋆], and Ramachandran Ramjee[⋆]
[⋆] Microsoft Research India; [†] Microsoft Research UK; [‡] USC; [§]University of Washington

ABSTRACT

The quest for higher data rates in WiFi is leading to the development of standards that make use of wide channels (e.g., 40MHz in 802.11n and 80MHz in 802.11ac). In this paper, we argue against this trend of using wider channels, and instead advocate that radios should communicate over multiple narrow channels for efficient and fair spectrum utilization. We propose WiFi-NC, a novel PHY-MAC design that allows radios to use WiFi over multiple narrow channels simultaneously. To enable WiFi-NC, we have developed the *compound radio*, a single wideband radio that exposes the abstraction of multiple narrow channel radios, each with independent transmission, reception and carrier sensing capabilities. The architecture of WiFi-NC makes it especially suitable for use in white spaces where free spectrum may be fragmented. Thus, we also develop a frequency band selection algorithm for WiFi-NC making it suitable for use in white spaces. WiFi-NC has been implemented on an FPGA-based software defined radio platform. Through real experiments and simulations, we demonstrate that WiFi-NC provides better efficiency and fairness in both common WiFi as well as future white space scenarios.

1. INTRODUCTION

Over the past decade, WiFi data rates have seen over a 100x increase. This was achieved through advances in physical layer wireless communication techniques (*e.g.*,OFDM, 64 QAM and MIMO) that provided increased spectral efficiency (bits/s/Hz). As further improvements in spectral efficiency become harder to achieve, using wider channels is being viewed as a solution to attain higher data rates. Today, 802.11n already allows for 40 MHz channels while the upcoming 802.11ac proposes 80 and 160 MHz channels.

In this paper we argue against this obvious approach of merely increasing the channel width to increase wireless data rates. Instead, we espouse the opposite – that the channels be no wider than existing 20 MHz WiFi channels and ideally be narrower, say 5 MHz or even 2 MHz. In order to achieve higher data rates then, unlike current day devices that operate over only one channel at a time, *we propose WiFi-NC a novel physical and MAC design that allows devices to run WiFi on several narrow channels, simultaneously and independently.* For example, a device must be able to use (transmit/receive on) eight 5 MHz channels instead of one 40 MHz channel. This diametrically opposite view is designed to address the following three key inefficiencies of current single (wide) channel systems (Section 3).

First, inefficiencies arise when heterogeneous radios coexist. While WiFi is designed to be fair to devices operating in the same channel, operation of 40 MHz devices near 20 MHz devices leads to starvation [1]. Consequently, 802.11n standard mandates devices to reduce their channel width to 20 MHz immediately upon detecting any coexisting 20 MHz device. As a result, in practical 802.11b/g/n deployments, 802.11n devices are often relegated to using only 20 MHz channels. To the best of our knowledge, no work has addressed this practical and common inefficiency in WiFi, which is bound to only get worse as 802.11ac devices with 80 MHz radios become available. In contrast, a 40/80 MHz WiFi-NC radio configured with two/four 20 MHz channels can make full use of its 40/80 MHz radio while still coexisting fairly with other 20/40 MHz networks.

Second, it is well-known that, due to MAC overheads such as backoffs, the increase in PHY data rates does not translate to commensurate gains in TCP/UDP throughput [22, 17]. To address this inefficiency, 802.11n standards support MAC-layer frame aggregation that allow frame sizes of up to 64KB, thereby reducing the relative impact of the MAC overhead. While frame aggregation works well for bulk data flows, other traffic such as TCP acks, VoIP packets and short HTTP flows are not amenable to such aggregation. The use of narrow channels in WiFi effectively also elongates packet transmission times relative to MAC overhead (for a given frame size, transmission time doubles when channel width is halved), thereby achieving higher throughput.

Third, fragmented spectrum can result in inefficient usage. For example, WhiteFi [4] uses variable width channels for operation in fragmented white spaces. However, the restriction of being able to use a single channel (wide or narrow) at a time, limits WhiteFi's ability to efficiently use free parts of the spectrum. For example, two 6 MHz narrow channels of a WiFi-NC radio can operate simultaneously on either side of a 6 MHz operating TV channel, while a single-channel system like WhiteFi will be restricted to choosing only one of the two bands. The ability to use multiple narrow channels simultaneously, allows us to devise an optimal throughput maximizing spectrum selection scheme, $TMax$, that is not possible in single channel systems.

Related work that comes closest to WiFi-NC is FICA [22]. FICA splits a single OFDM physical channel into narrower *sub-channels* and allows different devices to access them. However, sub-channels differ from narrow channels in a very fundamental way – in FICA, a new transmission opportunity arises only when the entire wide channel is idle; then, transmissions by devices over different sub-channels are tightly time synchronized ($\approx\ 10\mu s$) Thus, FICA is essentially a wide single-channel system with sub-channels that are interdependent. While FICA addresses the MAC inefficiency issue, the lack of independence among sub-channels precludes FICA from solving the inefficiencies due to heterogeneous radio co-existence or fragmented spectrum.

The centerpiece of WiFi-NC is the *compound radio*, a novel design, that uses a *single physical wideband radio but provides the abstraction of several independent narrow band radios – radiolets*, to the MAC layer. Each radiolet allows for independent carrier sensing, transmission and reception of packets in its own narrow channel. Radiolets are entirely implemented as digital circuits and provide the low cost and form factor benefits of digital processing.

The fundamental challenge in designing a compound radio is enabling efficient *interference isolation* among the radiolets – a compound radio must be able to simultaneously carrier sense, receive, and transmit over different radio-lets without any inter-dependence. Note that, even if conventional radios supported full duplex communication [7, 12, 19] over wide bands, we cannot simply divide the full duplex wide channel into multiple full duplex narrow channels and independently transmit/receive over these narrow channels. This is because while a full duplex radio will cancel out the spectral leakage of the narrow channel OFDM transmission at the *transmitter*, the spectral leakage can still cause severe degradation in adjacent narrow channels at the *receiver* (since the narrow channels are not synchronized). Thus, in order to achieve channel isolation, the compound radio uses *sharp elliptic filters* at both the transmitter and receiver (Section 5). These filters allows us to use very narrow guard bands between the radiolets (100KHz in our implementation), thereby paying an overhead of only 5% or 2% for a 2 MHz or 5 MHz narrow channel, respectively.

Another fundamental effect of using narrow channels is the need for *preamble dilation*. Since narrow channels transmit information at a slower rate, PHY layer preambles take longer to transmit. While the longer preamble results in only a small overhead for WiFi-NC (because data transmission times also dilate), a bigger issue is if this dilation results in increased carrier sensing time – in this case, WiFi-NC would need larger slots, which will severely affect channel utilization [17]. In order to avoid this problem, the compound radio uses energy detection to ensure carrier sensing time in WiFi-NC stays the same as in WiFi while employing cross-correlation over the dilated preamble in parallel for OFDM frame synchronization and frequency offset estimation.

We have prototyped the compound radio and WiFi-NC on a FPGA-based software defined radio platform. Through both real experiments on our testbed (Section 8) as well as extensive simulations (Section 9), we show that WiFi-NC is both more efficient and fair than WiFi. Further, while operating in white spaces, we show that WiFi-NC is able to achieve up to 121% higher throughput than WhiteFi [4] in the presence of background transmitters.

While the use of narrow channels has significant efficiency benefits, the primary cost is increased logic/memory requirements both at the transmitter/receiver (e.g., transmit and receiver filters, decoding logic per narrow channel, etc.). However, as the FPGA/ASIC sizes grow benefiting from Moore's law, we believe that the additional logic/memory requirements will not pose a significant constraint.

In summary, our paper makes three key contributions:

- The simple insight that radios with multiple independent narrow channels instead of a single wide channel can improve the efficiency of WiFi in many practical settings such as heterogenous radio co-existence, at high PHY speeds and operation in fragmented white spaces.
- WiFi-NC, a novel PHY-MAC design that operates WiFi independently over multiple narrow channels, and its implementation in the form of a compound radio on a FPGA-based software defined radio platform.
- $TMax$ algorithm for maximizing throughput by optimal frequency selection for WiFi-NC radios operating in white spaces.

2. RELATED WORK

There has been tremendous amount of work targeted towards improving WiFi and wireless communication. We discuss a few papers that are most relevant to WiFi-NC here.

Performance. A number of papers [13, 15, 20, 22] have proposed novel techniques to improve WiFi performance.

FICA [22] is closest to WiFi-NC in terms of advocating for fine-grain access. However, FICA proposes the use of *subchannels* for fine-grain access which is fundamentally different from the narrow channels of WiFi-NC. Subchannels in FICA require a *synchronous* system, where all nodes in carrier sense range must transmit within a few microseconds of each other. While it may be possible to time/frequenc synchronize all APs under one management, FICA will not perform well in practical settings where WiFi APs from several autonomous systems (businesses/homes) co-exist and are not time/frequency synchronized. Furthermore, even witl time/frequency sychronization, FICA does not tackle the inefficiencies that arise due to radios with different channel widths or operation in fragmented spectrum.

Coexistence. SWIFT [21] tackles the problem of coexistence of wide band radios in the presence of narrow band devices. The SWIFT radio detects narrow band transmissions and then weaves together the unused (non-contiguous) bands into one wireless link by transmitting only on the unoccupied frequencies. While both WiFi-NC and SWIFT support non-contiguous operation, SWIFT still uses the entire available, and potentially wide, band as a single channel, thus, suffering the same inefficiencies as WiFi.

Variable Channel Width. The use of 5 or 10 MHz channels can increase range and reduce power consumption [5]. However, the channel width adaptation in previous work [5] only configures the radio to *one* of 5, 10, 20 or 40MHz channel for a *single* communicating pair of radios. Coexistence/fairness will be an issue if multiple networks are configured with different channel widths. In WiFi-NC, each radio can choose to use one or more independent narrow channels, thus, gaining the benefits of narrow channels without sacrificing coexistence.

Guard bands. A number of techniques to mitigate the problem of adjacent channel interference was studied in [11]. The authors found that the use of guard bands was the most efficient solution to the problem. The issue of appropriate size for the design of guard bands was considered in [25]. The authors show that the size of guard band needs to be adapted based on the location of the wireless nodes. However, the software digital filters used in [24, 25] were Hamming window filters that do not have the sharp cutoff properties of the elliptic filters used in WiFi-NC (Section 5). Thus, we are able to show that even a small fixed guard band is conservative enough for our needs. Moreover, a system like Ganache [25] can also help adapt the guard band size in WiFi-NC dynamically.

Full-duplex. Recently, full-duplex single channel wireless communication systems have been proposed [7, 12, 19]. The key challenge in these systems is eliminating the self-interference of the local transmitter. Note that, if these systems operate over the standard 20MHz WiFi channel, they would also suffer the same MAC overhead inefficiencies as WiFi. Full-duplex communication is an orthogonal feature to WiFi-NC and can be added to the narrow channels of WiFi-NC.

Fairness. 802.11-based wireless networks exhibit unfairness due to a number of reasons including hidden terminals [8], capture effect [16], exponential backoffs (short term unfairness) [9], etc. WiFi-NC is focused on the problem of unfairness that arises when two or more networks operate over frequency bands that overlap (Section 3).

Overlapping Channels. Authors in [3] show significant unfairness in chaotic WiFi deployments where WiFi channels of adjacent access points can overlap and argue that better channel allocation and power control can help improve efficiency and fairness. Similarly, authors in [18] propose a frequency hopping algorithm called MAXchop for avoiding unfairness in uncoordinated deployments. Compared to these approaches, the narrow channel model of WiFi-NC reduces the possibility of partial overlap in channels.

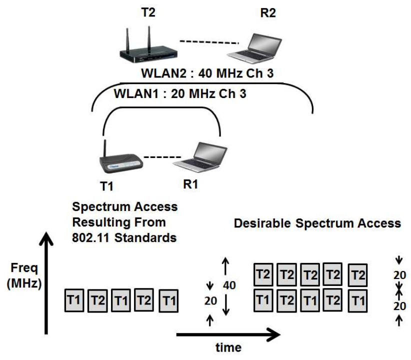
White spaces. Closest to WiFi-NC is WhiteFi [4], a WiFi-like system for TV white spaces. WhiteFi includes a spectrum assignment algorithm that maximizes a multichannel airtime metric called MCham and an algorithm called SIFT for detecting APs of varying channel widths. While WhiteFi supports variable channel width access, WhiteFi only supports contiguous operation over the channel. As we shall see in Section 7, the contiguous access restriction results in efficiency loss due to coexistence as well as due to the conservative behavior of the MCham metric. Since WiFi-NC supports independent narrow channels, non-contiguous operation through suppression of one or more narrow channels provides significant efficiency benefits when operating in fragmented white space spectrum. Authors in [24] propose Jello, a per-session FDMA system for latency sensitive applications such as streaming media. The focus is on utilizing (non-contiguous) white space spectrum over session durations rather than on a per-packet basis as in WiFi-NC. In addition, Jello does not consider fairness among distributed nodes, a key feature of WiFi-NC.

3. MOTIVATION FOR WIFI-NC

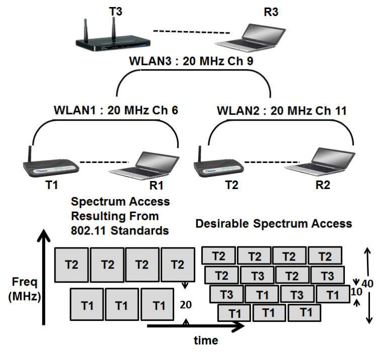
Existing wideband radios are monolithic, and access the channel in an "all-or-none" fashion. This design is inefficient in multiple settings and also leads to unfair channel access. We highlight these inefficiency and unfairness issues by using three examples.

Example 1 - Heterogeneous Radios: Inefficiency in Frequency. Consider the concurrent operation of two WLANs (Figure 1a) – an 802.11g WLAN1 (transmitter T1, receiver R1) operating on 20 MHz channel 3 and an 802.11n WLAN2 (T2, R2) operating on 40 MHz channel 3. As dictated by the 802.11n standard, the 802.11n radio detects the 20MHz transmitter and reconfigures itself to only operate on the 20MHz channel 3 (In fact, we were unable to get our 802.11n radio to operate in 40MHz mode in any of 2.4GHz channels in our lab due to this reason). Thus, transmitters T1 and T2 alternatively use the 20MHz band while 20MHz of frequency remains completely unused.

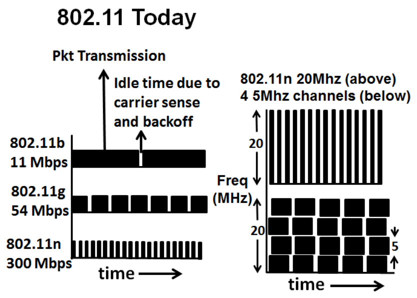
The above 802.11n mandate was a result of the observation that operation of 40 MHz 802.11n pre-standard devices (not subject to the above mandate) alongside 20 MHz 802.11g devices led to extreme unfairness and even starvation. To understand the reason for this unfairness consider the concurrent operation of three WLANs (Figure 1b) - WLAN1 (transmitter T1, receiver R1), operating on 20MHz Channel 6; WLAN2 (T2, R2) operating on 20 MHz channel 11; and WLAN3 (T3, R3) operating on 20 MHz channel 9. Since T3 is able to carrier sense both T1 and T2, 802.11 based CSMA dictates that it must wait until *both* T1 and T2 are not transmitting. However, T1 and T2 do not interfere with each other and may transmit whenever T3 is not transmitting. As depicted in Figure 1b, whenever T1 finishes transmitting a packet, T2 is still transmitting and vice-versa. Thus, T3 never finds its channel free for transmission result-


T2
R2
WLAN2 : 40 MHz Ch 3
WLAN1 : 20 MHz Ch 3
T1
R1
Spectrum Access Resulting From 802.11 Standards
Desirable Spectrum Access
Freq (MHz)
time

(a) Example I

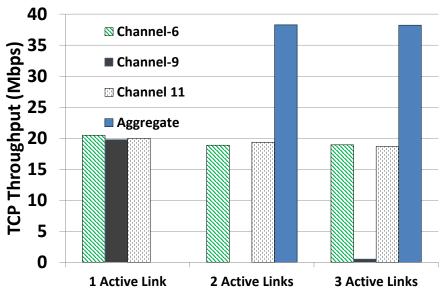


(b) Example I (partial overlapping)



(c) Example II

1: Examples



2: **Unfairness due to partial channel overlap**

ing in its starvation.

The above starvation effect also manifests itself when WiFi devices operate over overlapping 20MHz channels. To demonstrate this effect in a practical setting, we setup three identical 802.11b/g Netgear APs inside a lab area, and we had one client associated with each AP. The APs are configured to operate in channels 6, 9 and 11. The Figure 2 shows average TCP throughput at the clients for three different settings: (1) only one client is downloading at any time, (2) two clients R1 and R2 on non-overlapping channels, 6 and 11, downloading, and (3) all three clients are simultaneously downloading. From (1) and (2), it is clear that TCP flows on channel 6 and 11 are independent and do not interfere with each other when operating simultaneously. However, when all three flows are active, the client on channel 9 receives almost negligible throughput (570Kbps) compared to the other two clients (20 Mbps each) as depicted in case (3).

Example 2 - MAC Overhead: Inefficiency in Time. Another source of well-known inefficiency [22], illustrated in Figure 1c, arises from the fact that as the device bandwidth increases, while the time to transmit packets becomes smaller, the MAC overheads such as carrier sense and backoffs remain constant. 802.11n attempts to combat this unfairness by allowing for aggregated frames up to 64KB in size but this requires delaying frames at the interface in order to aggregate a large number of smaller packets and is not suitable for applications such as VOIP or short HTTP transactions. *An alternate approach that increases efficiency but does not require larger frame sizes is simply the use of narrow channels*. As seen in the Figure, reducing channel width from 20MHz to 5MHz simply results in quadrupling of the packet transmission time (the MAC overheads don't change). Thus, by elongating packet transmission time, narrow channels are able to better amortize the MAC time overheads.

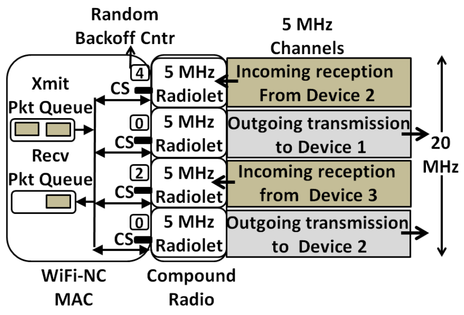
Note that there is a new inefficiency introduced due to narrow channels, namely, the guard band or the gap between two 5MHz channels. We show how this overhead can be kept very small (2% for 5MHz channels) in Section 5.

Example 3 - Fragmented Spectrum: The recent FCC ruling of T.V white spaces allows secondary devices to transmit in parts of the spectrum unoccupied by primary transmitters such as T.V broadcasts operating over 6 MHz channels. Such an opportunistic scenario often requires devices to operate in a fragmented spectrum. Consequently, a white space device with 40 MHz radio bandwidth may not find even a single continuous span of 40 MHz. A device that allows independent channel access and communication over several narrow channels (say, eight 5 MHz channels) will allow the use of fragmented spectrum more efficiently since the white space device can simply transmit around any occupied T.V. Channels. *This example shows that the future white space devices need to support non-contiguous operation which comes naturally to a device that has multiple independent narrow channels.*

4. WIFI-NC

In section 3 we saw that devices can achieve fairness, increased efficiency and can potentially better use fragmented spectrum if they used multiple independent narrow channels instead of a single wide channel. Given that WiFi already provides fair access to devices operating in the same channel (narrow or wide) through CSMA and backoff, WiFi-NC simply allows devices to operate WiFi independently over multiple narrow channels.

Figure 3 shows an illustration of WiFi-NC node configured with four 5MHz narrow channels using a 20MHz radio. The WiFi-NC MAC maintains independent random backoff



Random Backoff Cntr
5 MHz Channels
Xmit Pkt Queue
Recv Pkt Queue
CS
4
0
2
0
5 MHz Radiolet
Incoming reception From Device 2
Outgoing transmission to Device 1
Incoming reception from Device 3
Outgoing transmission to Device 2
20 MHz
WiFi-NC MAC
Compound Radio

3: WiFi NC implements WiFi over several narrow channels

counters and performs carrier sensing on each narrow channel. Whenever the backoff counter expires for a given narrow channel, a packet from the transmit queue is transmitted over the corresponding narrow channel. Similarly, packets can be received independently over narrow channels and placed in the receive queue. As can be seen from the Figure, the narrow channels are completely independent from each other. Thus, transmissions can be on-going simultaneously to different receivers (e.g., transmission to device 1 and 2), while other narrow channels can be in reception or carrier sensing mode. As we show in our evaluations (Section 9), this key property of independent narrow channels help WiFi-NC significantly outperform WiFi in terms of both efficiency and fairness in many common scenarios.

4.1 Exploring Design Choices

Off-the shelf radios allow operation on only one channel at time. In order to implement WiFi-NC there are several different alternatives. In this section, we consider these alternatives.

Use multiple narrowband radios on the same device. Several papers have advocated the use of multiple radios on a single node for better performance [2, 14]. Thus, one could consider implementing WiFi-NC using multiple narrow band radios. However, apart from several practical shortcomings such as cost, form factor, etc., there is also a fundamental drawback with such an approach – isolation requirement in the form of large guard bands between radios. For example, WiFi radios use a guardband of 3 MHz between two adjacent channels for interference free operation. This means that in order to create a compound radio of 80MHz with four 20 MHz radios, one would require guardbands worth 15 MHz (three 3 MHz in between the four radios and two on either side) - reducing spectral efficiency to 80%. This loss of spectral efficiency is further exacerbated as one uses narrower channels, say, 5MHz wide.

Use the sub-carrier structure of OFDM itself to enable fine-grain access. Prior work such as FICA [22] suggests that different nodes may use different sub-carriers within the same underlying physical channel to create sub-channels. However, in this approach actions such as Clear Channel Assessment (CCA), transmission and reception performed by different devices across all sub-channels must be tightly time synchronized. This is because in OFDM, sub-carriers overlap with each other and their accurate spacing and time synchronization is key to enable decoding at the receiver. Consequently, independent CCA, transmission and reception over sub-channels is not possible and leads to the same inefficiencies in co-existence between narrow and wideband devices described in Section 3.

Our Approach - Compound Radio. In order to enable multiple narrow channels, we propose a novel PHY-MAC design – the *compound radio*. The compound radio, *while using a single wideband physical radio device, performs digital processing to provide the abstraction of multiple independent radios to the MAC layer.* This is achieved by performing channelization digitally through digital filters and digital mixers as described in Section 5. Since digital filters allow for extremely cheap and high performance filters in comparison to analogue filters, digitally implemented adjacent channels require “very thin” guard bands (100 KHz in our implementation). Further, unlike overlapping sub-carriers in OFDM, these channels are completely separate from each other and have absolutely no cross-talk among them, allowing complete independent operation.

5. COMPOUND RADIO ARCHITECTURE

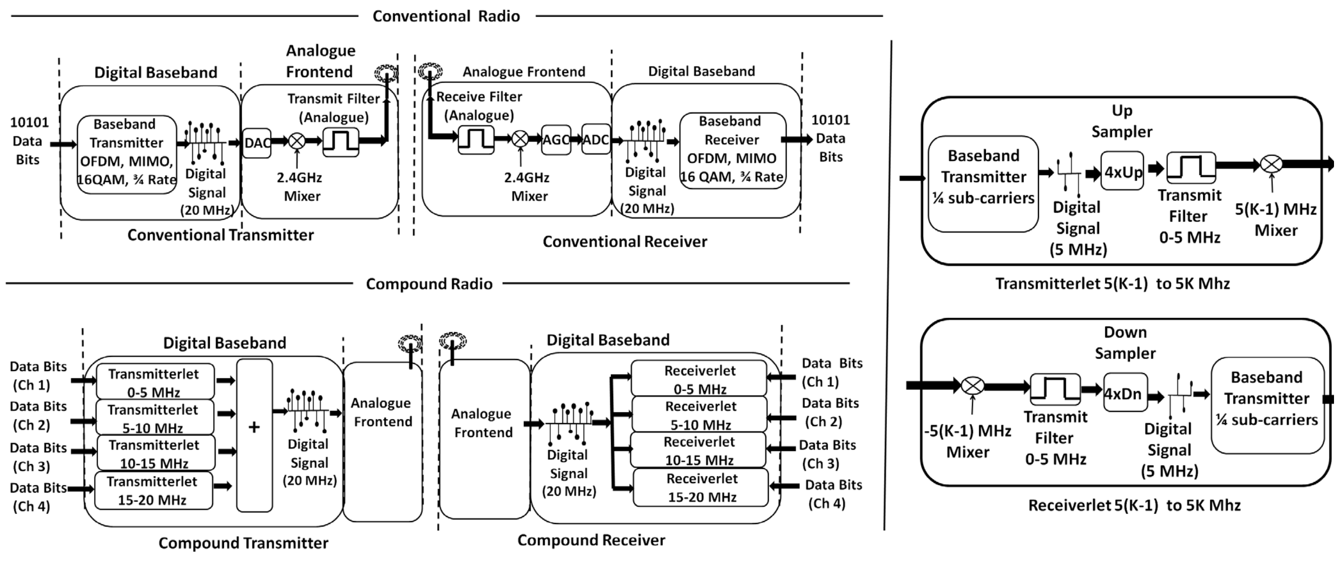
As discussed in Section 4, the compound radio provides an abstraction of multiple narrow-band radios while using only a single physical wideband radio. In this section, we start by describing the functioning of a conventional OFDM radio focusing only on the components that are necessary for providing the required background and then follow with our proposed architecture for the compound radio.

5.1 A Conventional Radio

As depicted in Figure 4 a typical radio transmitter or receiver consists of two key parts - an analogue front end and the digital baseband. Almost all the complex physical layer packet processing such as MIMO, OFDM, encoding and decoding etc. are implemented in the digital baseband since digital circuits provide the benefits of low cost, form factor and ease of implementation. However, as it is hard to design cheap digital circuits at clock rates of several GHz, the signal must first be down-converted from the carrier frequency (2.4/5 GHz) to the baseband frequency (20 MHz in case of 20 MHz channels) using the analogue frontend.

Analogue Transmit and Receive Filters. In order to avoid interference from/to devices operating over adjacent channels, radios use transmit and receive filters in the analogue front end (Figure 4). These filters, only let frequencies within the bandwidth of the channel to pass through (say 2.4-2.42 GHz for 20 MHz channel 1).

Mixer. The mixer, at the receiver, is responsible for down-converting the received signal at carrier frequency (2.4 GHz) to baseband frequencies (0-20 MHz) to be presented to the digital baseband. At the transmitter, it up-converts the baseband signal to carrier frequency making it suitable for trans-



4: Conventional Radio and Compound Radio

mission.

ADC and DAC. These are used to convert between the analogue signal at baseband frequencies to digital signal at the receiver and vice-versa at the transmitter.

AGC. Typical DAC circuits are designed to operate correctly for a specific input voltage range (say 0.5V to -0.5V). Thus, Automatic Gain Control (AGC) appropriately scales the analogue signal from the antenna to ensure that the signal from the antenna is within the desired voltage range.

Baseband transmitter/recevier. Generation and reception of packet including MIMO, OFDM, encoding, decoding, modulation and demodulation are handled by the baseband transmitter and receiver using digital circuits.

5.2 A Compound Radio

The key idea behind the compound radio architecture is to use digital mixers and transmit/receive filters in the baseband to create narrow channels digitally. Figure 4 depicts this idea for a compound radio that implements four 5 MHz narrow channels.

5.2.1 Compound Transmitter

The compound transmitter comprises N *transmitterlets*, each responsible for transmitting data over one narrow channel of width $\frac{B}{N}$, where B is the bandwidth of the analogue front end. Each transmitterlet consists of a baseband transmitter, an upsampler, a digital low pass filter and a digital mixer. The outputs of each of the transmitterlets are then added digitally and passed on to the analogue frontend which is identical to the analogue frontend of a conventional radio.

Baseband Transmitter. The baseband transmitter is identical to the baseband transmitter used in any conventional radio, except for two differences. First, since it operates over a channel that is $\frac{1}{N}$ the bandwidth, it uses $\frac{1}{N}$ number of subcarriers intended for the wide band channel. Second, it operates at $\frac{1}{N}$ the sampling frequency of that used for the wideband radio, since the required Nyquist sampling rate for the narrow channel is $\frac{1}{N}$ of that for the wide channel with bandwidth B. As discussed later in this section, this allows individual transmitterlets to operate at $\frac{1}{N}$ the clock rate and hence keep the total number of operations required per second across the N transmitterlets the same as the wide band radio.

Upsampler. In order to match the sampling rate of the wide band radio, the digital samples from the baseband transmitter are upsampled by a factor of N. During upsampling, $N-1$ additional digital samples are inserted between two consecutive samples through interpolation. There are several ways to interpolate – in our implementation, we use a DFT based upsampler.

Low Pass Filter. A sharp low pass filter (described in detail later in the section), ensures that the signal is indeed limited to within 0 to $\frac{B}{N}$ MHz.

Mixer. Prior to the mixer, the digital signals in all transmitterlets have frequencies between 0 - $\frac{B}{N}$ MHz. The digital mixer for the k^{th} transmitterlet shifts these frequencies by $\frac{(k-1)B}{N}$ MHz to ensure that the digital signal emanating from it has frequencies in the range $(\frac{(k-1)B}{N}, \frac{kB}{N})$ MHz. The mixer essentially multiplies each digital sample by a complex sinusoid of frequency $\frac{(k-1)B}{N}$ MHz and can be cheaply implemented using a ROM and two digital multipliers.

5.2.2 Compound Receiver

The compound receiver architecture is symmetric to that of a compound transmitter and consists of multiple *receiverlets* - each to receive packets over one narrow channel. Each receiverlet comprises, a mixer, a low pass filter, a down sampler and finally the baseband receiver. The mixer of the k^{th} receiverlet downs shifts the frequency of the received signal by $\frac{(k-1)B}{N}$ MHz. This frequency downshifting ensures that frequencies corresponding to the k^{th} receiverlet *i.e.,* in the range $(\frac{(k-1)B}{N}, \frac{kB}{N})$ MHz are mapped to the range $(0, \frac{B}{N})$ MHz. A low pass filter between $(0, B)$ MHz then extracts on the band corresponding to the receiverlet. A $\frac{1}{N}$ down-sampler then reduces the sampling rate by a factor of $\frac{1}{N}$ by simply dropping $N-1$ consecutive samples after picking each sample. The baseband receiver is identical to the baseband receiver of a conventional radio except that it operates at $\frac{1}{N}$ the sampling rate and uses $\frac{1}{N}$ sub-carriers of that used for the wideband channel.

6. DESIGN CHALLENGES

We faced two fundamental challenges in the design of the compound radio.

Interference Isolation. In WiFi-NC, nodes must be able to carrier sense, receive and transmit simultaneously on adjacent narrow channels. Since we use OFDM in each narrow channel for efficiency, the leakage from OFDM transmissions into the adjacent narrow channels can be significant (Section 6.1). We need to be able to isolate this interference within each narrow channel.

Preamble Dilation. While the use of narrower channels increases efficiency in WiFi-NC, channel widths below 20 MHz can lead to inefficiencies from increase in physical layer preamble lengths since narrow channels inherently transmit information at a lower rate.

In the rest of this section, we describe in detail each of these challenges and the approach we use to address them.

6.1 Interference Isolation

Figure 5 depicts a possible scenario with three WiFi-NC nodes. Node A simultaneously transmits to nodes B and C over narrow channels 1 and 3. At the same time nodes B and C transmit to node A over narrow channels 2 and 4 respectively. The spectrum of each of these transmissions as seen at Node A is also depicted in Figure 5 – node A's receiver experiences very high interference from its own transmissions in narrow channels 1 and 3 (about -20 dBm at the receive antenna, assuming a transmit power of 20 dBm [12]). Node B and C are located far away and their signals at A are extremely weak, about -85 dBm and -80 dBm, respectively.

Let as assume that we limit guard bands between narrow channels to 100 KHz so that even for a 2 MHz narrow channel, spectral wastage is only 5%. Figure 5 also depicts the typical OFDM spectral leakage in the absence of filters. The power in the adjacent channels decays to approximately about -40 dBm in the adjacent channels. In order to provide perfect interference isolation, the transmit and receive filters must attenuate the OFDM spectral leakage to below noise levels (-90 dBm or lower). Thus, we require an attenuation of the transmit signal by least 50dB to provide interference isolation. Thus, in our implementation, we use *transmit and receive filters that provide an attenuation of about 60 dB within 100kHz.* Note that this represents an extreme scenario for WiFi-NC where the self-interference is maximum compared to the received signal.

Filter Type	Bandwidth 5 MHz		Bandwidth 2 MHz	
	# Adds	# Mults	# Adds	# Mults
Chebyshev	76	76	48	48
Butterworth	492	492	208	208
Elliptic	26	20	22	17

1: Filter Comparison - 60dB attenuation, 100KHz guardband

Table 1 shows the number of adders and multipliers required to achieve our design (100 KHz guardband, 60 dB attenuation) by different choices of filters. As indicated in Table 1, Elliptic Filters [10] satisfy our requirements with the least number of elements. Consequently, we use Elliptic filters in our implementation.

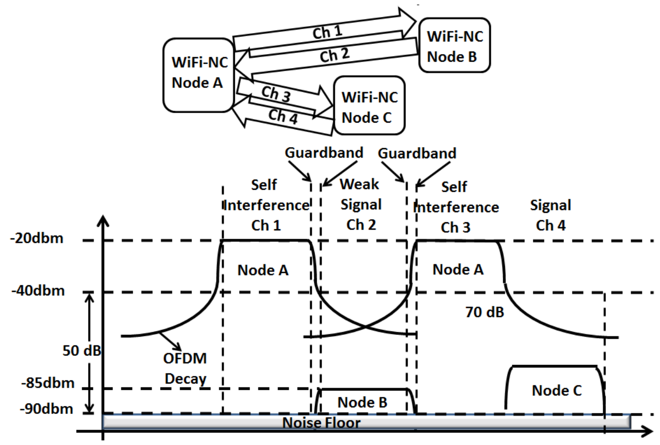
We have implemented the compound radio on an FPGA based software defined radio platform (Section 8). Figure 6 depicts the transmitted spectrum measured at a distance of 1cm from the transmit antenna for a 16QAM, 3/4 coding (36 Mbps) OFDM transmission over a 5 MHz narrow channel. As seen from Figure 6, the spectral leakage due to OFDM is significantly high and decays to only about -60dBm even at a distance of 2MHz from the transmitted band. The figure also shows the spectrum when using our transmit filter. We can see that the spectrum decays to about -90dBm beyond the 100 KHz guard band.

6.1.1 Effect of Carrier Frequency Offsets

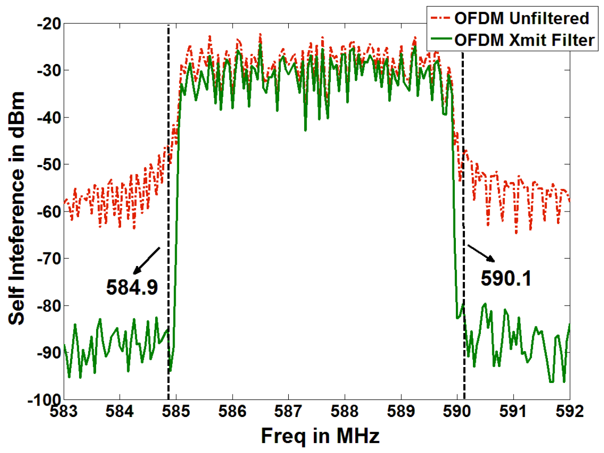
Due to manufacturing variations, two different radios invariably have a *carrier frequency offset* (CFO) - small difference in their carrier frequencies. These differences imply that the channel boundaries of two communicating radios will not be exactly aligned. This misalignment places a practical lower limit on how narrow guardbands can be in WiFi-NC since using guardbands smaller than the CFO will lead to interference leakage into adjacent narrow channels. The 802.11 standard requires CFO for any two pair of radios to be under 25ppm (60 KHz) in the 2.4 GHz band and under 20ppm (116 KHz) in the 5.8 GHz band. CFO for white space devices operating in the 200-800 MHz will be under 20 KHz (assuming 25ppm). Our choice of 100KHz guardband, thus, accommodates CFO in both white spaces as well as at 2.4 GHz in 802.11. In the 5.8 GHz band, however, a slightly larger guardband, perhaps 150 KHz wide, maybe required. In practice, since manufacturers typically ensure that the CFO is safely below the maximum allowed limit, we believe that CFO will not be an issue for WiFi-NC.

6.1.2 Effect of limited bits in ADC

While one can achieve self-interference isolation only by using sharp filters, this is possible only if the ADC of the



5: A possible scenario in WiFi-NC



6: Spectrum of transmission with and without filters

radio is able to support a wide range of power levels. An ADC typically accepts as input an analogue signal that is within ±0.5 V (or a similar range). Consequently, received signal is typically scaled (by a gain controller) down (or up) to lie within this range. The range of an ADC is specified in bits. Each extra bit of the ADC allows for discerning signals with half the amplitude and hence one fourth the power – in other words, each bit provides 6 dB resolution.

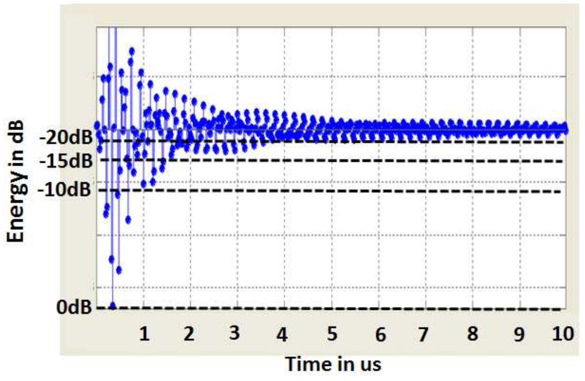
Since our testbed platform uses 14 bit ADCs, it has a range of 84 dB which means the radio is sensitive to signals that are 84 dB below the strongest received signal. In the face of -20dBm self interference then, a weak signal that is -85dBm effects only the last three to four bits of the ADC but is still discernible. However, many commercial systems use ADCs with 9 to 12 bits. Thus, for an ADC with 10 bits (60 dB range), this signal cannot be discerned at all since the last bit corresponds to -80dBm.

In devices with fewer ADC bits, analogue self-interference cancelation [19] or signal-inversion using Balun transformer [12] can be used to reduce the strength of self-interference so that power levels are in the range of the ADC. For example even a reduction of self-interference power by 25dB provided by Quellan QHx220 noise cancelers used in [19] or the 45dB over 40 MHz provided by Balun transformers [12] will permit devices with 9 bit ADCs to receive weak signals at -85 dBm while transmitting on adjacent channels.

Note that this cancelation is distinct from cancelation needed in full duplex systems [7, 19] – the cancelation here merely helps bring the power levels within the range of ADC so that transmit and receive can operate on *separate* channels while full duplex systems require cancelation of the full transmit signal so that one can receive on the *same* channel.

6.1.3 Filter Induced Interference

A filter restricts the spectrum of the signal by spreading (smoothing) it in time. The sharper the filter, the more the spreading. Figure 7 depicts the impulse response of our filter *i.e.,* the transmitted signal resulting from passing a single digital sample through the filter. As seen from the figure, the filter spreads the sample for several microseconds in time. This spreading in fact is the same effect as spreading due to indoor multipath environments. Such spreading results in self-interference between symbols termed Inter Symbol Interference (ISI).



7: Filter induced multipath

Need for longer Cyclic prefix (CP). In order to combat ISI, OFDM uses the cyclic prefix, which pre-appends 25% of the OFDM symbol and extends the symbol. The spread version of the previous symbol, thus interferes with the cyclic prefix and does not adversely effect the original symbol. Typical spreading due to multipath in indoor environments is less than 800 ns, consequently, WiFi uses 800 ns cyclic prefix. However, use of sharp filters in the compound radio increases this spreading. As seen in Figure 7, the spreading decays by about 10dB within 800 ns and to about 15 dB within 1.6μs. While low data rate modulations such as BPSK require about 6 dB SNR, higher data rate modulations such as 16 QAM may require up to 14 dB SNR. In our implementation we found that a cyclic prefix about 1.6μs long allowed for successful reception even at higher data rate modulation such as 16 QAM.

Increasing Number of Subcarriers. Cyclic prefix is a waste ful part of the transmission and results in a decrease in spectral efficiency. In order to keep efficiency the same after extending the CP by a factor η, the symbol duration must also be stretched by the same factor η. In OFDM this is typically achieved by increasing the number of subcarriers by a factor of η. In our implementation we found that $\eta = 2$ was sufficient to combat the impulse response of the sharp 60 dB filters. Consequently, while WiFi uses 64 subcarriers in a 20 MHz band, WiFi-NC must use 128 subcarriers.

6.2 PHY Preamble Dilation for channel widths below 20 MHz

Physical layer preambles are crucial for packet reception and perform several key functions. Since channels narrower than 20 MHz result in slower transmission of information compared to WiFi, it would take longer to transmit WiFi's physical layer preambles in each narrow channel of WiFi-NC. In this section we describe the effects of this *preamble dilation* and describe techniques to address them.

6.2.1 Preamble Dilation in WiFi-NC

The WiFi preamble can be divided into two logical parts - the *pre-synchronization* and the *post-synchronization*. Let us look at the functions of these two parts:

Pre-synchronization Preamble. This part of the preamble is primarily responsible for three important functions. *C*lear Channel Assessment (CCA) to sense if carrier is idle, *O*FDM frame synchronization to detect OFDM symbol boundary for decoding and *f*requency offset estimation to correct for mismatches in carrier frequency between transmitter and receiver. WiFi uses Pseudo-random Noise (PN) sequences to perform these three functions. The performance of a PN sequence depends on the length (number of PN samples) of the sequence. A narrow channel that has $\frac{1}{N}$ the bandwidth will take N times longer to transmit the same number of samples. Consequently, *this part of the preamble dilates by a factor of N, where N is the number of radiolets.*

Post-synchronization Preamble. After the receiver is synchronized to the transmitter, it must estimate and compensate for the distortions caused by the wireless medium. To aid this, the transmitter sends training symbols, one (or more) for each OFDM subcarrier. The receiver then estimates the differences between the received and expected symbols and corrects for them. The key observation here is that *the number of training symbols is proportional to the number of subcarriers.* Thus, while a narrow channel with $\frac{1}{N}$ the bandwidth transmits N times slower, the number of sub-carriers and hence the number of training symbols to be transmitted is also $\frac{1}{N}$ times lesser. Estimation of MIMO parameters, is based on a similar approach and is also proportional to the number of subcarriers. Consequently, since WiFi-NC uses 128 subcarriers instead of 64 used in WiFi (to counter the filter-induced interference), *this part of the preamble doubles in duration but is independent of channel width.*

How much does the preamble dilate for WiFi-NC? The pre-synchronization preamble in WiFi is 4μs while the post synchronization preamble varies between 4 OFDM symbols (16μs in 802.11g) to 9 OFDM symbols (16μs in 802.11n). Thus, for a WiFi-NC transmitter with N radiolets, the duration of the preamble transmission will be $4N + 32$ μs (802.11g) to $4N + 72$ μs (802.11n).

Preamble dilation can potentially affect the performance of WiFi-NC in two ways. First, it can reduce the efficiency since dilated preambles take longer to transmit and second, it can mean requiring larger slot durations than 9 μs. We examine each of these next.

6.2.2 Effect of preamble dilation on efficiency

While the preamble transmission duration increases, so does the duration to transmit data. For example, for WiFi-NC with a 2 MHz narrow channel, the 802.11n 300 Mbps preamble will dilate from 40 μs to 112 μs. At the same time, the time to transmit a 1500-byte packet elongates from 40 μs to 400 μs. Thus, *the ratio of preamble transmission time to packet transmission time still reduces significantly from 100% to 28%*, resulting in significant overall gain in efficiency as the 20 MHz WiFi channel is reduced to a 2 MHz narrow channel in WiFi-NC.

6.2.3 Effect of preamble dilation on slot duration

The slot duration of WiFi is fixed to be 9μs, 4μs of which are allocated to perform CCA, 1μs allows for propagation delays and 5μs for switching from receive to transmit mode. Since WiFi nodes use the pre-synchronization part of the preamble to perform CCA, dilation of this part of the preamble implies that slots might also need to be dilated since CCA must be performed within one slot duration. There are two different approaches to tackle this problem.

Decoupling slot width from preamble detection time. As used in WiFi-Nano [17], using interference cancelation and speculative preambles, one can decouple slot width from preamble detection time. This decoupling allows backoff slots to remain unchanged despite the preamble detection times being longer, thereby, preserving the backoff efficiency of WiFi.

Energy-based CCA. An alternative is to perform energy-based detection on the first 4μs of the dilated preamble for CCA rather than using preamble detection on the entire pre-synchronization preamble. This again decouples slot width from the other functions of the preamble such as frame synchronization and frequency offset estimation (which are performed in parallel), thereby, preserving the backoff efficiency of WiFi.

In this paper, we focus on the energy-based CCA approach. Fundamentally any CCA scheme must distinguish between receiver noise and harmful external interference, so as to avoid wasteful transmissions in the face of interference from other RF sources. In our implementation, in order to deem the channel as busy, the receiver collects samples over a 4µs sliding window (160 samples) and compares whether the sum of their squares (the energy) is larger than a threshold. To calibrate the threshold for each device, we first collect several tens of thousands of receiver noise samples in the absence of external transmissions. The threshold is then chosen to be the maximum collected energy over any 4μs window. While one could choose higher values of threshold (e.g. 10 dB higher), our choice of threshold is the most conservative – it ensures detection of *any* external interference. Being more conservative than WiFi can result in more backoffs than WiFi in a practical setting (e.g., due to microwaves). Since our experiments were conducted in the 580 MHz band (Section 8), we were not affected by the choice of this threshold.

7. WIFI-NC IN WHITE SPACES

In this section we consider the operation of WiFi-NC in white spaces. The key difference between operation in white spaces from that in the ISM band (2.4GHz) is that white space devices must avoid parts of spectrum occupied by primary users such as TV transmissions that use 6MHz wide channels. This leads to two key requirements for white space devices. First, they must be able to operate on fragmented spectrum *i.e.,* no single continuous span of spectrum as wide as 40 MHz or even 20 MHz may be available. Second, devices need to judiciously pick which parts of the spectrum to transmit on, given that several other white space devices may be operating – the *spectrum selection problem*.

The use of narrow channels allows WiFi-NC to efficiently use even narrow intermittent spaces between spectrum sections occupied by primary transmitters. Further, as we shall describe in this section, the ability to use multiple independent channels allows for a greedy distributed algorithm – $TMax$ that maximizes the total expected network throughput across all operating devices.

Prior Approach - WhiteFi. The problem of spectrum selection has been examined in WhiteFi [4]. WhiteFi allows the flexibility to select among three possible analogue frontend bandwidths 5 MHz, 10 MHz and 20MHz. While, the ability to use narrower bandwidths allows WhiteFi to operate over 5 MHz or 10 MHz even when there is no span of continuous 20 MHz spectrum available, WhiteFi devices may use only one channel at time. The authors propose a metric called $MCham$ that each device maximizes greedily to determine the center frequency and bandwidth of operation. $MCham$ metric for a node k with a certain center frequency f and front-end bandwidth B is given by

$$MCham_k(f, B) = \frac{B}{5} \prod_{c \in (f,B)} \rho_k(c) \tag{1}$$

Here, c corresponds to the 5MHz channels contained in the frequency span $(f - \frac{B}{2}, f + \frac{B}{2})$, and $\rho_k(c)$ corresponds to the expected share of node k in a 5MHz channel c, given by,

$$\rho_k(c) = \max\left(R_k(c), \frac{1}{L_k^c}\right). \tag{2}$$

In equation 2, $R_k(c)$ refers to the fraction of residual airtime available in the channel c and L_k^c refers to the total number of contenders in the channel.

WhiteFi was faced with a key constraint, i.e., its radio only supported the notion of a *single channel* that operated in a contiguous manner over the full bandwidth. This creates *two key disadvantages:* 1) the need to choose an operating bandwidth (e.g., 5MHz) that may be lower than the full bandwidth of the radio (e.g., 20MHz); 2) the $MCham$ metric has to be *conservative* since a wideband radio cannot use the channel until all overlapping subchannels are free at the same time leading to the *product term* in Equation 1. Note that this coupling could also result in starvation, similar to the problem described in Section 3.

$TMax$ **Algorithm in WiFi-NC.** In the case of WiFi-NC, since the radio supports *independent narrow channels*, both disadvantages of WhiteFi disappear. The radio can always use its full available bandwidth since it can operate in a non-contiguous manner around any primary transmitters. Also, since the narrow channels are independent, the available throu put estimate need not be conservative and is simply the *summation* of throughput in each of its narrow channels. Thus, WiFi-NC uses a new metric called Throughput Maximal metric or $TMax$ for determining its frequency of operation, as

$$TMax_k(f) = \sum_{c \in (f)} \frac{B}{n} \rho_k(c) \tag{3}$$

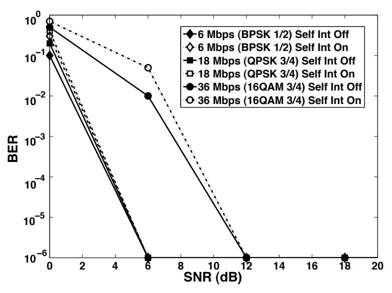
where n is number of narrow channels, B is the analogue frontend's bandwidth. and f is the set of all narrow channels in the range $(f - \frac{B}{2}, f + \frac{B}{2})$.

WiFi-NC nodes operating in white spaces, periodically scan over the entire available parts of the spectrum computing the $TMax$ metric for part. They then greedily choose f where the part of spectrum that maximizes $TMax$. When two or more regions of the spectrum have the same value for $TMax$, ties are broken by always choosing the lower frequency value for operation.

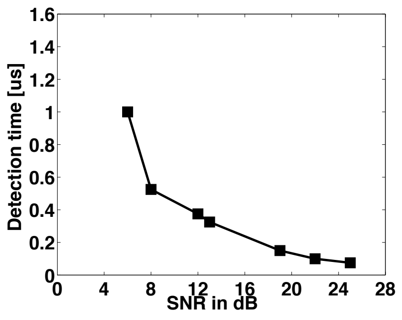
Optimality of $TMax$**.** It can be shown that, while each node greedily uses the $TMax$ algorithm, the scheme iteratively converges to maximize the expected aggregate network throughput across all operating devices and hence the overall spectrum utilization. In the interest of space, we do not provide the proof in this paper. A detailed proof can be found in [6].

8. RESULTS ON TEST BED

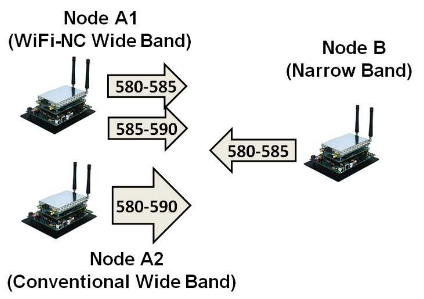
WiFi-NC has been implemented on a DSP/FPGA based software defined radio platform – the SFF SDR from Lyretech Inc. SFF SDR uses two Virtex-4 SX35 FPGAs and the DM6446 DSP processor from TI. The entire digital baseband of the compound radio and time-sensitive parts of MAC such as backoff counters and CSMA have been implemented on the FPGA. We used an off-the-shelf sub-gigahertz analogue radio front end provided by Lyretech that allows transmissions between 360 MHz to 960 MHz. The analogue radio front



8: Self-Interference Isolation



9: Performance of energy-based CCA



10: Experimental setup narrow band wide band fairness

end supports two antennas – one for transmitting and one for receiving – each of which can be operated independently. While, the board itself supports two different bandwidths namely 10 MHz and 20 MHz, throughout our experiments we have used the 10 MHz option. Using our implementation of the compound radio we demonstrate that WiFi-NC allows devices to share the spectrum in a fair and efficient manner.

8.1 Self-Interference Isolation

In this experiment we ask the question "how well can a WiFi-NC device receive while transmitting simultaneously on an adjacent narrow channel?" Specifically, *we demonstrate that there is no difference in BER for WiFi-NC, whether or not it transmits over an adjacent channel over a wide range of SNRs and data rates* indicating perfect self interference isolation. To answer this question we conducted an experiment with two WiFi-NC devices A and B as depicted in Figure 8. Node A transmits to Node B over the 5 MHz channel 585-590 MHz, while simultaneously Node B transmits to Node A over the channel 580-585 MHz. We measured the Bit Error Rates (BER) at Node A for various average SNR values generated by placing nodes at various distances. We then compared these values when only Node B transmits to Node A *i.e.,* in the absence of self-interference. We found that for data rates *up to 18 Mbps (QPSK with 3/4 coding rate) we were not able to see any bit error over* 10^6 *bits with or without self-interference even at SNRs as low as 5dB and at narrow channel widths of 2 and 5 MHz.*

16 QAM and higher data rate modulations however, are more sensitive to SNR. Consequently, in order to investigate at these higher data rates we tried 36 Mbps (16 QAM, 3/4 coding rate). As seen from Figure 8, for 36 Mbps (16QAM 3/4 coding rate) require about 14dB for the same. *This performance is almost identical when there is not self-interference indicating that the channels are isolated from self-interference leakage from the adjacent channel.*

8.2 Efficacy of Energy-based CCA

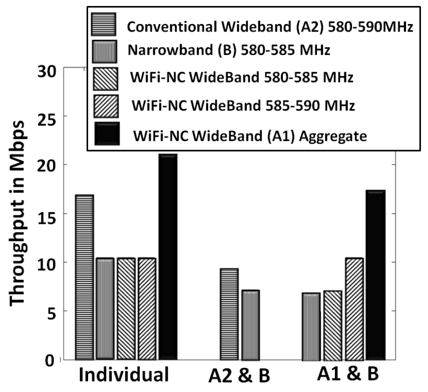
As discussed in Section 6.2.3, in order to enable CCA in 4 μs, we use an energy detection based scheme for channels narrower than 20 MHz. In this section we evaluate the efficacy of our energy-based CCA. The 802.11 standard demands a missed detection rate of 10%. *In our evaluation we asked the question, "At what window size of collected samples does the missed detection rate fall below 5%?".* In order to answer this question, a transmitter-receiver pair were placed at various distances from each other and for each distance the transmitter transmitted 1000 packets to the receiver. For each distance, we considered several different window sizes for collecting samples and performed CCA using each window size. The window size that gave us less than 5% missed detection rate was deemed as the detection time. *As seen from Figure 9, even signals with SNR as low as 5 dB are detected in about 1μs, while at high SNR values CCA can be performed in a few hundred nano-seconds.* This is expected since, the higher the SNR, the more easily signal can be distinguished from noise. Finally, we did not experience any false detection (detecting a non-existent transmission).

8.3 Narrow and Wide Band Device Coexistence

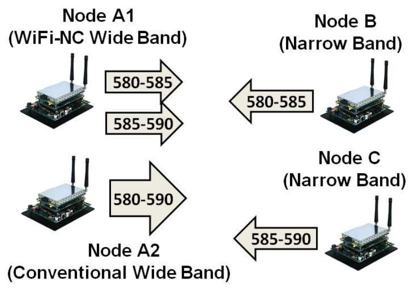
In this experiment we demonstrate that WiFi-NC allows narrow and wide band devices to coexist in a fair manner. The experimental setup is depicted in Figure 10. As shown in Figure 10, Nodes A1 and A2 are wideband devices operating over 10MHz while Node B is a narrow band device operating over a 5 MHz channel. Node A1 is a WiFi-NC device and uses two narrow 5 MHz channels while node A2 is a conventional device and uses a single 10 MHz wide channel.

Individual Links. First, for measuring the base achieved throughputs, we measure the throughputs achieved by each device on each narrow channel individually, with- out any other transmissions. All devices are operated at 54Mbps. As seen from Figure 11, the 10MHz wideband device achieves a throughput of about 16Mbps, while the achieved throughout over each 5 MHz narrow channel is about 10 Mbps.

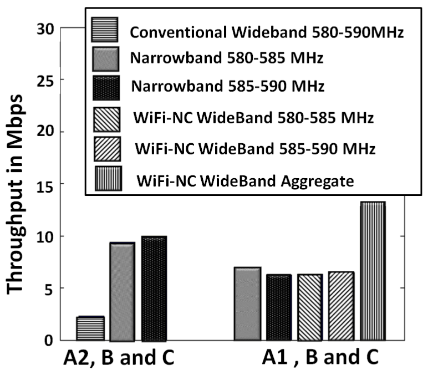
Conventional 10 MHz and Narrow Band. Next, (A2 & B in Figure 11) device A2 and B are turned on to start transmitting. Since A2 uses a wide channel it shares the channel with B and vice-versa. Consequently, while A2 achieves roughly



11: Narrow band wide band coexistence



12: Experimental setup for starvation



13: WiFi-NC solves starvation

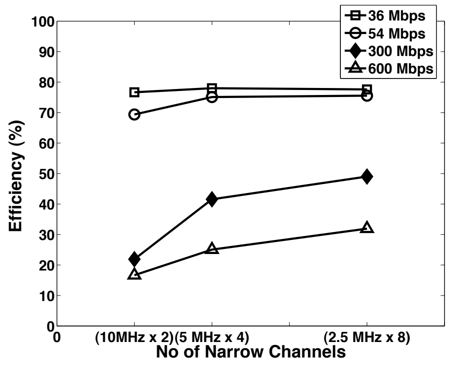
9 Mbps of throughput, B achieves about 6 Mbps throughput.
WiFi-NC 10 MHz and Narrow Band 5 MHz. Finally, A1 is turned on and operated alongside B. As seen from Figure 11 (A1 & B), A1 only shares the 5 MHz channel between 580-585 MHz with B and is able to completely use the channel 585-590 MHz without any contention. Consequently, A1 is able to achieve an aggregate throughput of about 15 Mbps while B achieves a throughput of about 6 MHz. *This demonstrates how WiFi-NC can help narrow and wideband devices gain fair access and thus keep the overall utilization high.*

8.4 WiFi-NC Avoids Possible Starvation

As discussed in Section 3, operation of devices with wide channels alongside those with narrow channels can result in starvation. In this experiment we demonstrate that WiFi-NC devices can avoid this starvation by using narrow channels. As depicted in Figure 12, there are four nodes used in this experiment. A 10 MHz wideband device, A2 operating over 580-590 MHz, a 10 MHz WiFi-NC device A1 using two narrow channels 5 MHz each and two narrowband devices B and C operating over non-overlapping 5 MHz bands 580-585 MHz and 585-590 MHz.
Conventional 10 MHz and two 5 MHz devices. We first, turn on devices A2, B and C which transmit packets while contending for channel access. As seen from Figure 13, node A2 achieves only about 2 Mbps out of a possible 16 Mbps (Figure 11) while devices B and C achieve most of the share in their respective bands. *This demonstrates how wideband devices can potentially suffer from extreme unfairness while operating alongside non-overlapping devices.*
WiFi-NC 10 MHz and two 5 MHz devices. Next, we turn off A2 and turn on A1 allowing the WiFi-NC wideband devices to transmit while contending for channel access. *As seen from Figure 13 the WiFi-NC device is able to share the narrow band channel fairly with each of the narrow band devices B and C and consequently avoid extreme unfairness.*

8.5 Efficiency in WiFi-NC

In order to measure efficiency gains in WiFi-NC with the use of narrower channels we implemented radiolets with 10,



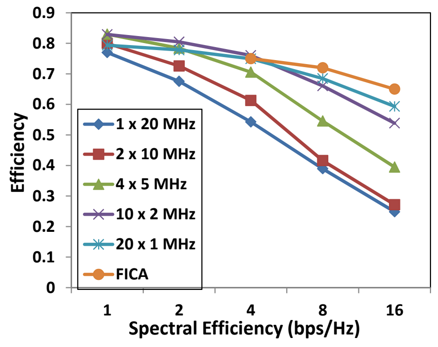
14: Increase in efficiency with number of narrow channels

5 and 2.5 MHz narrow channels on our platform. Since we did not have a MIMO implementation, in order to create the effects of various data rates we used shorter packets with data in the packet scaled by the data rate. For example, to create the effect of 300 Mbps we used 36 Mbps with packets of 1500(36/300) bytes. Figure 14 shows the variation of efficiency with narrower channels. As depicted in Figure 14, efficiency increases with increase in increasing number of narrow channels and as expected, the increase is greater at higher data rates such as 300 and 600 Mbps.

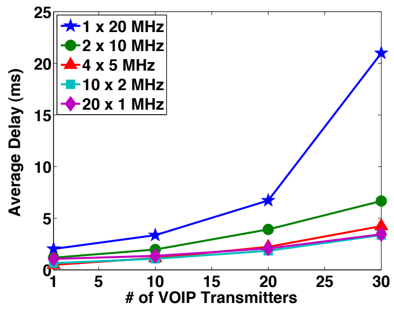
9. SIMULATION STUDY

The testbed evaluation is restricted to small scale experiments involving a few devices and limited set of scenarios. Several questions regarding the performance of WiFi-NC require exploration. How do the choices of different channel width effect the efficiency of WiFi-NC? How does WiFi-NC perform with latency-sensitive media traffic such as VOIP compared to WiFi? How does WiFi-NC perform as a potential choice for white space usage? In order to answer these questions, we have implemented the compound radio PHY layer and WiFi-NC MAC layer as extensions to the Qualnet network simulator.

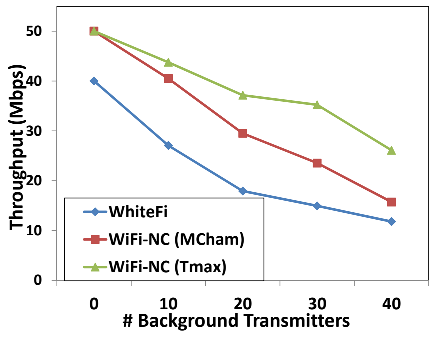
In our simulations, all nodes are within carrier sense range of each other. Unless otherwise noted, we use a spectral effi-



15: Efficiency for different channel sizes and spectral efficiencies



16: Average Latency for different channel sizes



17: White space transmitter throughput

ciency value of 2.7 bps/Hz and a radio front-end bandwidth of 20MHz, which is equivalent to the 54 Mbps mode of 802.11a/g. Protocol overhead such as header size and ACK length is modeled on 802.11a. For all WiFi-NC configurations, we fixed the guard band size to 100 KHz, same as our prototype implementation.

9.1 Efficiency

To understand the trade-off of using smaller narrow channels, we experimented on a single 20 MHz wide-band link with different WiFi-NC configurations (number of channels $\times$ channel width). We measure the achieved bit rate when the link is saturated with 1500 byte back-to-back packets for different values of spectral efficiency (bps/Hz). Figure 15 shows the channel efficiency, which is computed as the ratio of the achieved bit-rate to the raw bit-rate, for different WiFi-NC configurations. For comparison, we also plot the channel efficiency numbers quoted by FICA [22].

As spectral efficiency increases, the fixed protocol overheads become increasingly burdensome and consequently, narrower channels provide much better channel efficiency than using a single wide channel. With a spectral efficiency of 16 bps/Hz, equivalent to a 320 Mbps bit rate across a 20 MHz band (similar to 300Mbps 802.11n), the 20 $times$ 1 MHz configuration is 60% efficient compared to only 25% when using a single 20 MHz band. In comparison, FICA achieves an efficiency of around 65%. Thus, WiFi-NC is able to match the high efficiency of a synchronous system like FICA while still operating in a fully asynchronous manner.

9.2 Latency

Narrower channels increase throughput by elongating packet transmission times; this amortizes the cost of fixed overheads. However, longer transmission times also increase transmission latency od each packet which could adversely affect latency sensitive traffic such as VOIP. Surprisingly, however, we found that using *narrower channels actually reduces system latency* when there are multiple clients.

In this experiment, along with a single bulk transmitter saturating the link, we add an increasing number of clients that each transmit 200B packets every 20 ms representing VOIP payload. Figure 16 shows that, as the number of clients increases, narrow channels have lower latency compared to a single 20MHz channel. As the number of clients increase so does contention and consequently the likelihood of collisions. Since latency sensitive clients do not saturate the channel, using narrower channels reduces the number of contenders on any given band and thus reduces latency. In particular, using more channels reduces the incidence of packet collisions due to choosing the same slot for transmissions by up to 10% (not shown due to lack of space).

9.3 White Space Networking

WiFi-NC is also well suited for use in white space networks where fragmented spectrum is the norm. To demonstrate this, we simulated a white space network based on TV broadcasters in an urban area according to TV Fool [23]. This gave us thirty one 6 MHz TV channels with 12 primary transmitters. In order to study the impact of background traffic, among the open channels we randomly distributed narrow-band background transmitters, each of which used a UDP stream to consume 1/3 the capacity of a single 6 MHz band, similar to the evaluation used in WhiteFi [4]. We then added a wide-band transmitter that can use up to 4 channels, and we measured its throughput for three different schemes, WhiteFi [4], WiFi-NC with the $MCham$ metric and WiFi-NC with the $TMax$ metric.

Figure 17 shows the throughput as the number of background transmitters increases. With 0 background transmitters, WhiteFi can use 4 channels concurrently and achieves a 40 Mbps throughput. However, as more background transmitters are added, $MCham$ is forced to select bands that use less than four channels to avoid background transmitters. This means that when there are 40 background transmitters, WhiteFi selects only a single channel, and achieves only 12 Mbps of throughput. In the case of WiFi-NC with $MCham$ metric, throughput is higher due to WiFi-NC's higher efficiency and also its ability to contend with each narrow band transmitter independently. With 20 background transmitters, this scheme increases throughput by more than 65% compared to WhiteFi. However, with 40 transmitters its gains reduce since the $MCham$ metric selects only one channel.

When WiFi-NC is used with the $TMax$ metric, $TMax$ always uses four adjacent channels and contends independently on each one (operating in a non-contiguous manner around incumbents). Thus, it is able to deliver a throughput gain of up to 121% over WhiteFi.

10. DISCUSSION

Three common concerns with any new wireless design are *backward compatibility*, *energy consumption*, and *implementation complexity*.

Backward Compatibility. Given that most WiFi devices today use 20MHz channels, supporting backward compatibility in WiFi-NC is easy. Upon detecting a 20 MHz transmission in its vicinity, the WiFi-NC radio simply reconfigures itself to use 20MHz wide channels. For a 80 MHz 802.11n WiFi-NC radio, using four 20 MHz narrow channels can provide fairness and efficiency gains while being compatible to legacy WiFi devices.

Energy Consumption. Since the efficiency of WiFi-NC is higher than that of standard WiFi, transmitting the same amount of information requires radios to be turned on for a lesser amount of time. A large component of the power consumed by an RF circuit can be attributed to the analogue components such as amplifiers, analogue filters and the oscillator. Since WiFi-NC does not require any changes to the analogue front end, this part of the power consumption is same as that for WiFi. Consequently, we believe that WiFi-NC will also be more power efficient than WiFi.

Implementation Complexity. Indeed, the implementation complexity of digital logic in WiFi-NC is higher than that of WiFi. However, digital circuits enjoy the scaling properties of Moore's law. For example, ASICs and FPGAs have seen a 300% increase in terms of number of gates in the past few years. Given this trend, we believe that accommodating the complexity of WiFi-NC on a chip will not be a significant deterrent in the adoption of WiFi-NC.

11. CONCLUSION

In order to support gigabit wireless speeds, 802.11 standards are increasingly being driven towards wide channel design. In this paper, we argue for supporting multiple independent narrow channels within a single wideband radio and propose WiFi-NC. To enable WiFi-NC, we propose a novel radio design, the compound radio. Through experiments and simulations, we show that WiFi-NC maintains high efficiency at high data rates and is able to fairly utilize the wideband in the presence of coexisting networks. Further, WiFi-NC is also well suited for future white space scenarios where spectrum may be fragmented.

12. ACNOWLEDGEMENTS

We thank our shepherd, Brad Karp, and the anonymous reviewers for their constructive comments.

13. REFERENCES

[1] 802.11n Bonding Unfairness. http://www.zdnet.com/blog/ou/80211n-draft-110-a-kinder-gentler-neighbor/410.
[2] A. Adya, P. Bahl, J. Padhye, A. Wolman, and L. Zhou. A multi-radio unification protocol for IEEE 802.11 wireless networks. In *BroadNets*, 2003.
[3] A. Akella, G. Judd, S. Seshan, and P. Steenkiste. Self management in chaotic wireless deployments. In *ACM MobiCom*, 2005.
[4] P. Bahl, R. Chandra, T. Moscibroda, R. Murty, and M. Welsh. White space networking with wi-fi like connectivity. *ACM SIGCOMM*, 2009.
[5] R. Chandra, R. Mahajan, T. Moscibroda, R. Raghavendra, and P. Bahl. A case for adapting channel width in wireless networks. In *ACM SIGCOMM*, 2008.
[6] K. Chintalapudi et. al. WiFi-NC: WiFi over Narrow Channels. Technical Report MSR-TR-2012-17, Microsoft Research, 2012.
[7] J. I. Choi, M. Jain, K. Srinivasan, P. Levis, and S. Katti. Achieving single channel, full duplex wireless communication. In *ACM MobiCom*, 2010.
[8] S. Gollakota and D. Katabi. Zigzag decoding: Combating hidden terminals in wireless networks. In *ACM SIGCOMM*, 2008.
[9] M. Heusse, F. Rousseau, R. Guillier, and A. Duda. Idle sense: An optimal access method for high throughput and fairness in rate diverse wireless lans. In *ACM SIGCOMM*, 2005.
[10] M. C. Horton and R. J. Wenzel. The Digital Elliptic Filter – A Compact Sharp-Cutoff Design for Wide Bandstop or Bandpass Requirements. *IEEE transactions on Microwave theory and techniques*, pages 307–314, May 1967.
[11] W. Hou, L. Yang, L. Zhang, X. Shan, and H. Zheng. Understanding cross-band interference in unsynchronized spectrum access. In *ACM CoRoNet*, 2009.
[12] M. Jain et. al. Practical, real-time, full duplex wireless. In *ACM MobiCom*, 2011.
[13] K. Jamieson and H. Balakrishnan. Ppr: Partial packet recovery for wireless networks. In *ACM SIGCOMM*, 2007.
[14] S. Kakumanu and R. Sivakumar. Glia: A practical solution for effective high datarate wifi-arrays. In *ACM MobiCom*, 2009.
[15] S. Katti, H. Rahul, W. Hu, D. Katabi, M. Medard, and J. Crowcroft. Xors in the air: Practical wireless network coding. In *ACM SIGCOMM*, 2006.
[16] A. Kochut, A. Vasan, A. Shankar, and A. Agrawala. Sniffing out the correct physical layer capture model in 802.11b. 2004.
[17] E. Magistretti et. al. WiFi-Nano: Reclaiming WiFi Efficiency through 800ns slots. In *ACM MobiCom*, 2011.
[18] A. Mishra, D. Agrawal, V. Shrivastava, S. Banerjee, and S. Ganguly. Distributed channel management in uncoordinated wireless environments. In *ACM MobiCom*, 2006.
[19] B. Radunovic, D. Gunawardena, P. Key, A. Proutiere, N. Singh, V. Balan, and G. Dejean. Rethinking indoor wireless mesh design: Low power, low frequency, full-duplex. In *WiMesh*, 2010.
[20] H. Rahul, F. Edalat, D. Katabi, and C. Sodini. Frequency-Aware Rate Adaptation and MAC Protocols. In *ACM MobiCom*, Beijing, China, September 2009.
[21] H. Rahul, N. Kushman, D. Katabi, C. Sodini, and F. Edalat. Learning to share: Narrowband-friendly wideband wireless networks. In *ACM SIGCOMM*, 2008.
[22] K. Tan, J. Fang, Y. Zhang, S. Chen, L. Shi, J. Zhang, and Y. Zhang. Fine Grained Channel Access in Wireless LAN. In *ACM SIGCOMM* August 2010.
[23] TV FOOL. http://www.tvfool.com.
[24] L. Yang, W. Hou, L. Cao, B. Y. Zhao, and H. Zheng. Supporting demanding wireless applications with frequency-agile radios. In *USENIX NSDI*, 2010.
[25] L. Yang, B. Y. Zhao, and H. Zheng. The spaces between us: Setting and maintaining boundaries in wireless spectrum access. In *ACM MobiCom*, 2010.

Catching Whales and Minnows using WiFiNet: Deconstructing Non-WiFi Interference using WiFi Hardware

Shravan Rayanchu, Ashish Patro, Suman Banerjee
University of Wisconsin Madison

Abstract

We present WiFiNet— a system to detect, localize, and quantify the interference impact of various non-WiFi interference sources on WiFi traffic using commodity WiFi hardware alone. While there are numerous specialized solutions today that can *detect* the presence of non-WiFi devices in the unlicensed spectrum, the unique aspects of WiFiNet are four-fold: First, WiFiNet quantifies the actual interference impact of each non-WiFi device on specific WLAN traffic in real-time, which can vary from being a *whale* — a device that currently causes a significant reduction in WiFi throughput — to being a *minnow* — a device that currently has minimal impact. WiFiNet continuously monitors changes in a device's impact that depend on many spatio-temporal factors. Second, it can accurately discern an individual device's impact in presence of multiple and simultaneously operating non-WiFi devices, even if the devices are of the exact same type. Third, it can pin-point the location of these non-WiFi interference sources in the physical space. Finally, and most importantly, WiFiNet meets all these objectives not by using sophisticated and high resolution spectrum sensors, but by using emerging off-the-shelf WiFi cards that provide coarse-grained energy samples per sub-carrier. Our deployment and evaluation of WiFiNet demonstrates its high accuracy — interference estimates are within $\pm 10\%$ of the ground truth and the median localization error is $\leq$ 4 meters. We believe a system such as WiFiNet can empower existing WiFi clients and APs to adapt against non-WiFi interference in ways that have not been possible before.

1 Introduction

WiFi devices share the unlicensed spectrum with a plethora of other devices and technologies. A few examples include Bluetooth headsets, ZigBee devices, cordless phones, various game controllers (Xbox, Wii, etc.), and custom wireless security camera systems. Even non-communicating appliances such as microwave ovens, leak energy into this spectrum. Each such device can cause interference to WiFi communication. Since WiFi's underlying standard (IEEE 802.11) does not have any explicit mechanism to recognize such non-WiFi sources of interference, typical WiFi links have no reasonable way to guard against such interference. In this paper, we design *WiFiNet* — a collaborative neighborhood of WiFi nodes — to "catch" various non-WiFi transmitters causing harmful interference to

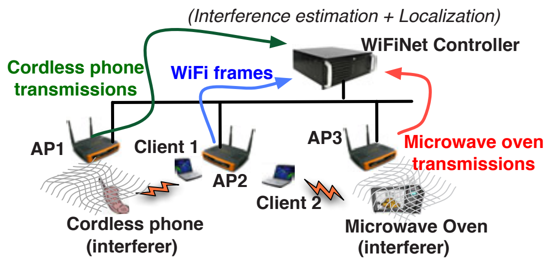

(Interference estimation + Localization)
WiFiNet Controller
Cordless phone transmissions
WiFi frames
AP1
Client 1
AP2
AP3
Client 2
Microwave oven transmissions
Cordless phone (interferer)
Microwave Oven (interferer)

Figure 1: Illustration of WiFiNet's architecture.

WiFi communication (Figure 1). More specifically, through WiFiNet we can answer the following questions — *how much* interference is any non-WiFi RF transmitter (e.g., a Bluetooth headset, an active analog phone, or a microwave oven) causing to an existing WiFi communication and *where* in the physical space is each such non-WiFi interferer located?

Much of the prior work has employed custom hardware to tackle non-WiFi interference. Examples include commercial products such as AirMaestro [1] and Wispy [4] that build specific signatures to *detect* the presence of a device. Recent research efforts (e.g., RFDump [13], DOF [8], TIMO [18]) have used the flexibility allowed by software radios to develop novel signal processing techniques and physical layer designs to co-exist with these devices. The unique aspect of WiFiNet is that it is built entirely on top of standard WiFi network interface cards (NICs). In particular, an emerging class of WiFi NICs, such as those based on the Atheros 9280 chipset, as part of their WiFi frame decoding process, provide coarse-grained energy samples per sub-carrier of a WiFi channel. These energy samples are a few orders of magnitude lower in resolution than those available to sophisticated spectrum analysis tools. In our recent work Airshark [16], we have shown that even with such a low resolution system, a regular WiFi node (either an Access Point or a client) can individually *detect* the presence of non-WiFi devices.

Airshark is, however, only the first step in the broad space of deconstructing non-WiFi interference and quantifying their impact on WiFi links. WiFiNet leverages collaboration between multiple WiFi nodes to address both quantification of interference impact and localization of these interferers, as we explain below.

Quantifying non-WiFi interference impact in real-time: The mere presence of a non-WiFi device, as detected by Airshark, in the vicinity of a WiFi transmitter is not always harmful. For instance, an active analog cordless phone at a specific location, may only have a minimal impact on a particular WiFi link. We call such a low-impact non-WiFi device, a *minnow.* On the other hand, a microwave oven radiating a significant amount of energy in its vicinity might cause severe disruption to nearby WiFi links. We call such an interferer, a *whale.*

However, the impact of interference from the same non-WiFi device can quickly change over time. For instance, if the microwave oven's setting is adjusted to operate with a low power level, this device may suddenly turn into a minnow. On the other hand, if the cordless phone user moves to a different location which is closer to the WiFi link, this device might turn into a whale with respect to this WiFi link. It is even possible that the impact of the cordless phone on the WiFi link changes due to properties of the WiFi link itself. For example, when the WiFi link is operating at 54 Mbps, the disruptive impact of the cordless phone is quite high, with the impact decreasing as a rate adaptation algorithm reduces the WiFi link's choice of PHY rates. WiFiNet tracks this continuously changing impact of non-WiFi transmitters on WiFi communication in real-time, adjusting its interference estimates immediately as operating parameters change (e.g., the microwave power setting is changed, or the WiFi device's PHY rate selection algorithm starts operating with a higher rate).

Locating non-WiFi interferers: WiFiNet also determines the physical location of such non-WiFi transmitters immediately, so that the precise source of such interference can be determined, and if needed, such interfering devices can either be re-configured or disabled.

Through these new and unique capabilities, WiFiNet provides new RF management tools for WiFi environments using off-the-shelf WiFi NICs only, obviating the need for sophisticated wireless hardware. In fact, WiFiNet can be easily implemented and integrated into enterprise WiFi APs to achieve improved mitigation strategies against non-WiFi interference for enterprise environments.

1.1 Challenges in designing WiFiNet

In designing and implementing the capabilities of WiFiNet, we had to overcome the following set of challenges:

How to detect multiple devices of the same type? In many wireless environments, there are multiple devices of a given type, e.g., two different cordless phones. It is possible that among these two phones, one is a whale and causes 80% loss in throughput to a WiFi link, while the other is a minnow and causes only 5% loss in throughput. To differentiate between these two interferers, WiFiNet needs to determine how many devices of each type are operating at any given instant. To achieve this goal, WiFiNet utilizes tight clock synchronization, and employs signal clustering techniques operating on some device specific attributes (when available) and signal strength observations gathered by multiple WiFi detectors to identify the unique transmission contributions from different, potentially identical, non-WiFi devices. Our prior work, Airshark, builds signatures of each device type to detect the presence of any such device in the vicinity of the detecting WiFi node. But such an individual WiFi node is not able to determine if there is only one or two or three different cordless phones in the vicinity, and hence, cannot attribute which part of wireless transmissions belong to which such interferer.

How to estimate each device's impact? After segregating each non-WiFi device's transmissions, WiFiNet uses fine-grained timing analysis for estimating the impact of each interferer — time-frequency overlaps between the WiFi frames and non-WiFi device's transmissions are analyzed and correlated with the outcomes (frame success or loss) to discern the impact of each device. Our technique works well for both low and high duty (duty of 100%) devices. In our design, we take into account the carrier sensing interference, interference from WiFi sources and multiple PHY rates of operation used by WiFi links.

How do we localize the non-WiFi device? Localization in indoor wireless environments is a well studied problem [5, 6, 20, 23]. Common techniques include signal strength based triangulation [23] and RF fingerprinting approaches [5]. However, the key requirement for such localization approaches is for multiple detectors to *detect the same transmission* at different signal strengths. In the commonly known WiFi localization techniques, this is easy because the different detectors decode the same wireless frame and use the frame's identity to ensure sameness.

In our case, the WiFi detectors cannot decode the non-WiFi transmissions, and hence cannot immediately assign the same identity to "pulses" received from the non-WiFi transmitters. A core challenge that we needed to solve is for different WiFi detectors to determine which received pulses correspond to a single transmission from the same non-WiFi device. The next challenge is to build a model for localization. Propagation characteristics are similar for both WiFi and non-WiFi transmitters since they operate on the same frequency. WiFiNet exploits this fact and builds the model by exchanging WiFi frames and recording signal strength measurements. Since the transmit power of non-WiFi devices can be arbitrarily different from that of WiFi nodes, the model takes this into account by operating on the *difference* in received signal strengths. Through experiments, we show the feasibility of this approach for non-WiFi device localization using WiFi-only detectors.

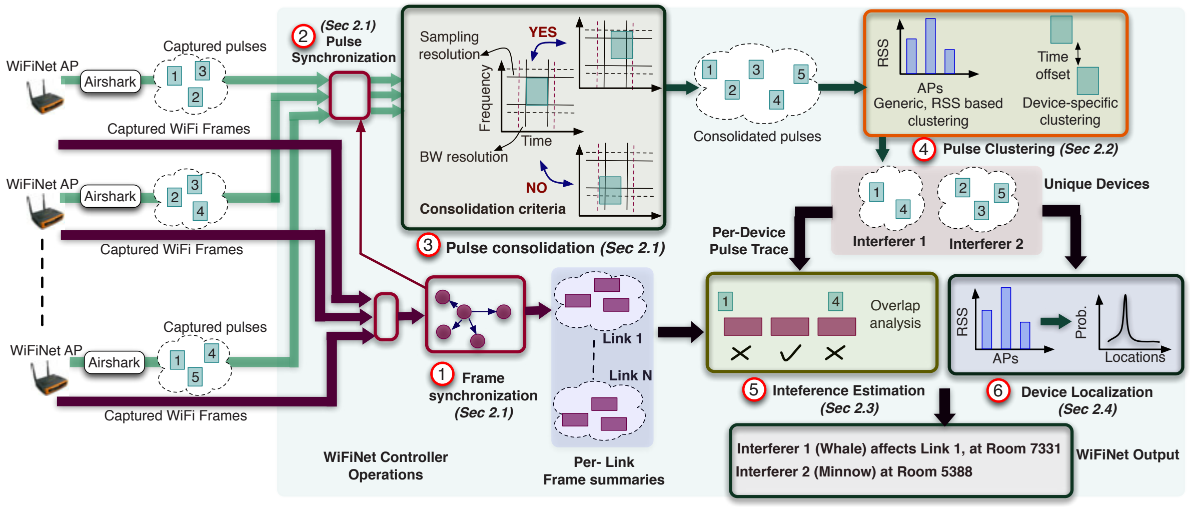

WiFiNet AP
Airshark
Captured pulses
Captured WiFi Frames
(Sec 2.1)
2 Pulse Synchronization
Sampling resolution
YES
NO
Frequency
Time
BW resolution
Consolidation criteria
3 Pulse consolidation (Sec 2.1)
Consolidated pulses
RSS
APs
Generic, RSS based clustering
Time offset
Device-specific clustering
4 Pulse Clustering (Sec 2.2)
Unique Devices
Interferer 1
Interferer 2
Per-Device Pulse Trace
1 Frame synchronization (Sec 2.1)
Link 1
Link N
Per- Link Frame summaries
Overlap analysis
5 Inteference Estimation (Sec 2.3)
Prob.
Locations
6 Device Localization (Sec 2.4)
WiFiNet Controller Operations
Interferer 1 (Whale) affects Link 1, at Room 7331
Interferer 2 (Minnow) at Room 5388
WiFiNet Output

Figure 2: Flow of operations in WiFiNet. WiFiNet APs capture spectral samples as well as WiFi frames. Each AP runs Airshark [16] to detect non-WiFi devices and output non-WiFi pulses (transmissions) tagged with device type. (1) WiFi frames are used to synchronize the clocks at the APs. (2,3) Synchronized clocks at the APs are then used to consolidate the pulses across multiple APs using a heuristic (§2.1). (4) Consolidated pulses are clustered using RSS and device-specific attributes to output unique non-WiFi device instances and their pulses (§2.2). (5) For each non-WiFi device instance and WiFi link, the interference estimation module analyzes the impact of the device on the link using transmission overlaps (§2.3). (6) Model-based localization algorithms are used to localize each non-WiFi device instance (§2.4).

Summary of key contributions: Summarizing, the key contributions of our WiFiNet system are three-fold: (i) it detects and discerns the transmission contributions of different non-WiFi interferers in the vicinity of the WiFi detectors; (ii) it attributes interference impact of each such non-WiFi device for any given WiFi link, classifying them as whales, minnows, or anything else in between, through collaborative observations; and (iii) it pinpoints the location of each such non-WiFi interferer so that they can be independently re-configured or disabled. All of these capabilities are implemented using WiFi-only detectors.

The entire WiFiNet system has been implemented using the Atheros AR 9280 based WiFi NICs, and evaluated in detail through various experiments. Our results indicate a typical impact determination accuracy of $> 90\%$ and a localization error of ≤ 4 meters in these environments.

2 WiFiNet

We start by presenting an overview of WiFiNet's architecture, followed by the details of its design and operation.
Architecture and flow of operations. WiFiNet employs collaborative observations from multiple WiFi-only detectors spread across a network to perform its non-WiFi device interference estimation and localization operations. Since most enterprise APs today come equipped with multiple WiFi radios, one way to deploy WiFiNet would be to employ one of the radios as a detector. In such a setting, WiFiNet can function as follows. All the enterprise APs are connected to a central controller over an Ethernet backplane. Each AP can have two radios: (i) a regular radio used to communicate with the clients, and (ii) a detector radio that continuously captures spectral samples as well as WiFi frames. APs run Airshark [16] to process the spectral samples and perform device detection. Post detection, Airshark outputs a set of "pulses" (time-frequency blocks representing non-WiFi device transmissions). Since WiFi hardware cannot decode non-WiFi pulses, Airshark can only provide *limited information* for each pulse — pulse's start and end timestamps, its center frequency and bandwidth, its average received power, and a tag that indicates its device type (e.g., Bluetooth). Next, APs also process the captured WiFi frames to create a per-client frame transmission summary: frame start and end timestamps, PHY rate, and reception status (*i.e.*, whether the AP received an ACK for this frame or not). The proximity between the two radios ensures that the detector radio receives the majority of frames transmitted by the regular radio due to capture effect, thereby creating an accurate summary of frame transmissions [21]. The per-client WiFi frame transmission summaries and the captured non-WiFi pulse traces are forwarded to the controller to identify the individual non-WiFi device instances, estimate their interference impact and localize them. Figure 2 presents the overall control flow. We now explain each of these tasks in detail.

2.1 Identifying unique pulses

Since the same pulse can be received by multiple APs in the WLAN, the first task for the controller is to consolidate the traces and identify the *unique pulses* transmitted by different non-WiFi devices operating in the environment. To do this, the controller has to identify the "common" pulses received by the APs and create a single consolidated pulse. However, finding common pulses is not straightforward as WiFi APs *cannot decode* non-WiFi pulses.

Pulse consolidation. WiFiNet uses a heuristic to consolidate the pulses: if two APs receive a pulse that has the same device type (e.g., Bluetooth), has the same start and end times, has the same center frequency and bandwidth, then most likely the APs received the same pulse (transmitted by a particular non-WiFi device). In practice, we allow a certain leeway as these parameters might not exactly match e.g., we allow the maximum difference between the pulse start (and end) times to be FFT sampling resolution of the WiFi card (116μs for AR9280 card) and that between pulse center frequencies (and bandwidths) to be resolution bandwidth of the WiFi card (312.5 kHz or equal to 802.11 sub-carrier spacing).

To apply the heuristic, however, would require the pulse traces at the APs to be synchronized. How do we synchronize the pulse traces without knowing common pulses (*i.e.*, reference points)? WiFiNet solves this issue by leveraging the WiFi hardware — the timestamps of the pulses are derived from the *same clock* that is used to timestamp the captured WiFi frames. WiFiNet first synchronizes the clocks at all the APs using captured "common" frames as reference points, and then uses the synchronized APs to find "common" pulses. We implement a graph-based, opportunistic synchronization approach similar to [22]: the controller first synchronizes "pairs of APs" using common reference frames, and then transitively synchronizes all APs. To account for the clock drift, the synchronization process is repeated every 100 ms, which results in tight synchronization between the APs (an error of < 4 μs). Since, the technique is completely passive, it doesn't generate any additional wireless traffic.

Output from consolidation. The controller applies the appropriate synchronization offsets to each AP's pulse trace and then finds the common pulses among the APs using the heuristic mentioned above. The consolidation process can be carried out efficiently as the pulses are sorted by time. After consolidation, the controller is left with unique pulses transmitted by non-WiFi devices, and for each unique pulse, we associate an RSS vector $\mathbf{r} = [r_0, \ldots, r_{N-1}]$ that represents the received power of this pulse at each of the N APs in the WLAN. We set r_i to the average received power of the pulse at ith AP, if the pulse was indeed received this AP, otherwise $r_i = \phi$.

2.2 Identifying unique device instances

After obtaining the unique pulses, the next task for the controller is to detect the number of non-WiFi device instances, segregate the pulses belonging to each instance and establish a unique ID for it. WiFiNet first segregates the pulses according to their device type, and employs *clustering algorithms* for further segregation. The algorithms determine "the number of clusters" (non-WiFi device instances), and assign each pulse to a cluster. The combination of (device type, cluster center) is then used as the ID for this device instance. In our current prototype, we implement (i) a generic, RSS based clustering that is applicable to all non-WiFi devices and (ii) clustering based on timing properties that is specific to some non-WiFi device types. We now explain both approaches.

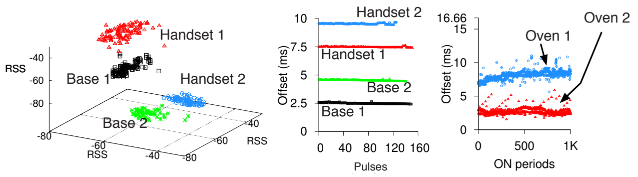


Figure 3: Segregating pulses in the presence of multiple, simultaneously operating devices of the *same type*,based on WiFiNet's device specific and generic clustering. Figure shows clusters of pulses from (left) 4 FHSS cordless devices using a generic, RSS based k-means + EM-clustering technique using 3 WiFiNet APs (middle) 2 FHSS cordless phone base/handset pairs (4 FHSS cordless devices) using pulse start time offset (right) 2 Microwave ovens using ON-period offset.

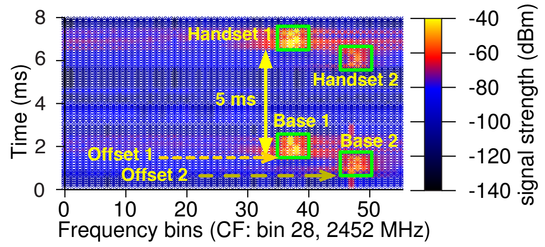


Figure 4: Heatmap of 4 FHSS cordless phone devices (2 base/handset pairs) captured by a WiFiNet AP, showing the timing property.

2.2.1 Generic clustering based on signal strength

WiFiNet's generic clustering approach operates on N-dimensional RSS vectors (*i.e.*, vector sizes grow with the number of APs). To improve the performance of clustering algorithms, we use some optimizations: (i) clustering is performed every *scan window* (5 secs in our current prototype) to keep the number of pulses low, (ii) dimensions corresponding to APs not receiving any pulse in the current window are discarded. Before proceeding with clustering, however, we have to tackle the another problem: some RSS vectors might have missing values (*i.e.*, $r_i = \phi$) for some columns. This is because APs might capture pulses intermittently (i) as they are far from the device, or (ii) due to a stronger signal from other WiFi or non-WiFi transmissions [16] that overlapped with the pulse. While it is possible to define a distance function for clustering that ignores missing values in the vectors, such a function is unsuitable for many traditional clustering algorithms as it doesn't satisfy certain mathematical properties such as the triangle inequality [10]. This presents us with two choices, (i) use clustering algorithms which allow a certain degree of freedom in the formulation of a suitable distance function or (ii) fill in the missing values using a best-effort approach, and then use traditional clustering algorithms. We explored both these choices.

Clustering algorithms. For approach (i), we explored density-based clustering, DBSCAN [12] that allowed us to use a distance function that ignores the missing values — distance between two pulses was calculated using a function that operates only on received signal strengths at "common" APs and ignores other APs whose RSS values are missing [19]. For approach (ii), a standard way is to use *imputation*, where missing values are replaced using "most likely" values. In WiFiNet, we use *EM-Imputation* [3], a well known imputation method, where the missing values are replaced by using expectation maximization with a multi-variate normal model. After imputation, we can use traditional clustering mechanisms as the distance function (e.g., Euclidean) can now operate on all the columns of the vectors. We experimented with several clustering algorithms and found that a combination of k-Means and EM-clustering perform the best: we iteratively run the k-Means clustering algorithm with different values of k ($1 \leq k \leq k_{max}$), and then pick the best solution [3]. A cross validation approach is used to pick the best k such that the 'within cluster sum of squared errors' is minimized. This is used as the initial solution to the EM-clustering algorithm, which outputs the final non-WiFi device instances and the corresponding pulses. In our experiments, we set $k_{max} = 10$, *i.e.*, we assume that the maximum number of simultaneously operating devices of the same type to be 10. Figure 3 (left) shows the result for RSS based clustering of 4 FHSS cordless phone devices using 3 WiFiNet APs. In §3, we evaluate both the clustering algorithms.

2.2.2 Clustering based on device specific attributes

We found that some non-WiFi device types exhibit certain specific timing properties that can be exploited to provide better clustering performance compared to the generic RSS based clustering approach.

— *Pulse start time offset for FHSS cordless phones.* WDCT cordless phone sets cycle through frames of 10 ms: each frame consists of two short pulses, one emitted by the base at the beginning of the frame and the other by the handset, occurring after 5 ms (both at the same center frequency). Both base and handset then jump to a different center frequency for the next frame. Figure 4 shows the pulses from two cordless phone sets (*i.e.*, 2 base/handset pairs, a total of 4 unique cordless phone devices) captured by WiFiNet. Figure 3 (middle) shows that clustering based on the pulse start time offsets (t MOD 10) can segregate the pulses belonging to each device.

— *ON-period offset for microwave ovens.* Microwave ovens emissions exhibit an ON-OFF pattern, typically periodic with a frequency of 60 Hz (frequency of the AC supply line) *i.e.*, a period of 16.66 ms [16]. WiFiNet computes the offset for start times of the microwave pulses (ON periods) as t MOD 16.66 and uses this to segregate their pulses. Figure 3 (right) shows the result of clustering pulses from two microwave ovens operating simultaneously.

The RSSI based generic clustering doesn't work very well when the multiple devices of the same type are placed close to each other. The device specific properties (based on timing properties) are not affected by the distance between multiple devices and thus improves clustering performance. But, these device specific properties are not available for all interferer types. Thus, WiFiNet uses both clustering mechanisms to identify the number of unique device instance.

2.3 Interference Estimation

After clustering, the WiFiNet controller has a set of clusters, each representing a unique non-WiFi device instance. We now explain how the controller can analyze the interference impact of each device instance.

Intuition and Overview. For each non-WiFi interferer instance and a WiFi link, the WiFiNet controller performs interference analysis by correlating the the link's frame transmission with the non-WiFi device's pulse transmissions and observing the reception status of the frames. WiFiNet measures the impact of a non-WiFi device on a WiFi link by computing the probability of a frame loss when the frame overlaps with a simultaneous transmission from the non-WiFi device. Intuitively, the extent of interference is directly proportional to the probability of losing overlapping frames. For instance, in Figure 5,frames F_1, F_2 and F_3 correspond to a WiFi link and pulses T_1 and T_2 belong to a nearby non-WiFi device. The controller observes that frames transmitted on the link are *unsuccessful* whenever the non-WiFi device's *pulse overlaps with the frame in time (and frequency) i.e.*, frames F_1 and F_3 overlap with non-WiFi device pulses T_1 and T_2, and are lost. It can therefore infer that the device strongly interferes with the link since an overlapping pulse always causes a packet loss. Such fine-grained timing analysis is possible because APs are tightly synchronized (§2.1) and they use the *same clock* to timestamp both pulses and the frames. We now explain our interference estimation metrics.

Metrics for interference estimation. Formally, the interference estimation metrics used in WiFiNet can be explained as follows. Let I be the event that interference from a particular non-WiFi device causes a frame transmission to be unsuccessful. Let L be the event of an unsuccessful transmission due to background losses (e.g., due to weak signal) and O denote the event of an overlap between the frame transmission and a simultaneous transmission (e.g., a pulse) from the non-WiFi interferer.

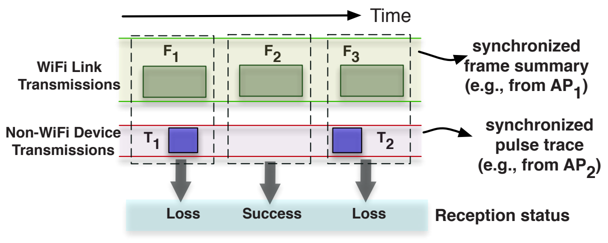

Time
F1
F2
F3
WiFi Link Transmissions
synchronized frame summary (e.g., from AP1)
Non-WiFi Device Transmissions
T1
T2
synchronized pulse trace (e.g., from AP2)
Loss
Success
Loss
Reception status

Figure 5: Illustration of interference estimation in WiFiNet.

— *(Metric 1) Impact given overlap.* Conditional probability, $p[I|O]$ is used to measure the *impact given overlap i.e.*, probability that a frame is unsuccessful given an overlap with a simultaneous transmission from a non-WiFi device.

— *(Metric 2) Overall impact.* WiFiNet also maintains $p[I]$, the *overall impact* of a non-WiFi device. Here, $p[I]$ is equal to $p[I|O]{\cdot}p[O]$ (when there is no overlap, $p[I|\neg O]$ is simply 0). That is, $p[I]$ is probability of frame loss due to the overall activity from the non-WiFi device.

We note that $p[I]$, the overall impact of the interferer, depends on the probability of overlap $p[O]$, which varies based on the link and interferer transmission patterns. Whereas, $p[I|O]$ is *not affected* by these transmission patterns *i.e.*, $p[I|O]$ indicates the *worst case impact* of the interferer on the link, which is observed when $p[O]{=}1$ (*i.e.*, when the transmissions of link and the interferer always happen to overlap). Next, we explain how these probabilities are estimated by WiFiNet in real-time.

Interference estimation. The controller measures the total number of frames transmitted (n) on the WiFi link of interest, the number of frames that overlapped with the non-WiFi device's transmissions (n_o) and n_o^l, the number of overlapped frames that were unsuccessful. It then computes $p[O]$, the probability of transmission overlap as n_o/n. Next, the controller computes $p[(I{\cup}L)|O] = n_o^l/n_o$ *i.e.*, the probability of an unsuccessful frame transmission due to either background losses or interference from the non-WiFi device, given an overlap in transmissions. It also computes the probability of frame loss when there is no overlap from the interferer, $p[L]$ as n_{no}^l/n_{no}. Here, $n_{no} = n{-}n_o$ is the number of frames without overlap and n_{no}^l is the number of n_{no} transmissions lost. Since L is independent of O, we have $p[L|O] = p[L|\neg O] = p[L]$. Also, I and L are independent events, and so we have $p[(I{\cup}L)|O] = p[I|O]{+}p[L]{-}p[I|O]{\cdot}p[L]$. That is,

$$p^{\text{WiFiNet}} = p[I|O] = \frac{\Big(p[(I{\cup}L)|O] - p[L]\Big)}{(1-p[L])} \qquad (1)$$

Using $p[(I{\cup}L)|O]$ and $p(L)$, the WiFiNet controller estimates $p[I|O]$. Following this, the controller also computes the overall interference $p[I]$ as $p[I|O]{\cdot}p[O]$.

Handling overlaps from multiple non-WiFi interferers. In general, a frame transmission may overlap with multiple simultaneous transmissions from potential non-WiFi interferers. In this case, the WiFiNet controller attributes the frame transmission success or loss to each overlapping non-WiFi interferer. We observed that *diversity* in the frame transmission times [21] as well as the diversity in transmission times of different non-WiFi devices allows WiFiNet to distinguish the *true* non-WiFi interferer from the other *false* non-WiFi interferers (*i.e.*, devices that happened to transmit at the same time as the true interferers). In particular, such a diversity allows WiFiNet to observe further transmissions from false non-WiFi interferers that overlap with the frames but do not lead to a frame loss. In our experience, such a transmission diversity arises due to (i) distinct transmission characteristics of different non-WiFi devices (e.g., frequency hopping devices typically emit short pulses at different center frequencies) and (ii) diversity in the usage times of non-WiFi devices [16], where in a typical enterprise not more than $3{-}4$ devices were found to be *simultaneously active*.

2.3.1 Enhancements to the basic technique

Handling high duty devices operating with other devices. Transmissions from multiple devices that *always* happen to overlap in time can lead to cases where WiFiNet can make incorrect estimates. For example, WiFiNet may identify a false interferer as a true interferer if the transmissions from the false interferer always happen to overlap with that of a true interferer. In our experiments, we found that such a scenario is unlikely when using pulsed transmitters (e.g., ZigBee devices) or frequency hopping devices (e.g., Bluetooth or FHSS cordless phones) that typically emit short pulses. However, operating high duty devices (e.g., analog cordless phones) that *continuously* emit energy alongside other low duty interferers will cause their transmissions to always overlap that can lead to incorrect estimates.

We use two refinements to the basic approach to correctly identify interference impact of a low duty interferer W operating alongside a high duty device H: (i) when computing $p[I_H|O_H]$ for a high duty device, we only consider the frames that *do not overlap* with a transmission from any other non-WiFi device. The background losses $p[L]$ are computed using packet losses when no interferers are present. (ii) For computing the $p[I_L|O_L]$ of low duty interferers operating in the presence of high-duty devices, the background losses ($p[L]$) are computed using using packet loss information when the low duty interferer is not present. These losses include the link propagation losses as well as the losses due to the high duty devices. Equation 1 is used to compute both $p[I_H|O_H]$ and

$p[I_L|O_L]$. We refer the reader to the technical report [19] for details of this mechanism.

Quantifying impact at different 802.11 rates. The impact of a non-WiFi interferer on a WiFi link also depends on the PHY rate being used by the WiFi transmitter. To account for this, the WiFiNet controller records the overlaps and losses separately for each different PHY rate, and computes a separate interference estimate for each rate. This helps quickly to estimate interference at each PHY rate when using dynamic bit rate adaptation as opposed to high-overhead bandwidth tests that require controlled experiments at each PHY rate to estimate the same [21].

Handling sender-side interference. Similar to the procedure used for interference analysis, the WiFiNet controller can infer whether a WiFi transmitter is deferring to a non-WiFi device by correlating the WiFi frame transmissions with the non-WiFi pulse transmissions. Two cases of interest arise. Case(a) when the WiFi transmitter is not deferring to the non-WiFi device, the WiFiNet controller will observe several instances where the frame transmission *starts while the pulse transmission is in progress*. Case(b) When the WiFi transmitter is indeed deferring to the non-WiFi device, the WiFiNet controller will not observe instances where frame transmission starts while the pulse transmission is in progress. However, this condition alone is not enough to infer that the WiFi transmitter is deferring to the non-WiFi device, as it may happen that the WiFi transmitter did not have any packets to send while the pulse transmission was in progress *i.e.*, the WiFi transmitter did not contend for the medium. To identify the deferral instances, we use a heuristic similar to the prior work on carrier sense estimation between WiFi links [15, 21]: the controller identifies the deferring frames as those where the difference between the pulse transmission end time and the frame transmission start time is within a certain threshold δ_w. Here, δ_w is the maximum time spent by the WiFi transmitter performing back-off and is set to $28+320$ μs (DIFS + Max back-off period for 802.11g).

The controller can now compute the fraction $\Delta_{cs} = \frac{n_d}{n_d+n_{nd}}$ where n_{nd} is the number of Case (a) instances that indicate *non-deferral* behavior and n_d is the number of Case (b) instances that indicate *deferral* behavior. If the transmitter is indeed deferring, Δ_{cs} would be close to 1. Whereas, if the transmitter is not deferring to the non-WiFi device, the difference in the pulse and frame start transmission times would be uniformly distributed in the interval $[0, \delta_p+\delta_w]$, where δ_p is the duration of the pulse. That is, we expect $\Delta_{cs} \approx \frac{\delta_w}{\delta_p+\delta_w}$. Typically, $\delta_p > \delta_w$, therefore Δ_{cs} is low for cases of non-deferral (e.g., for δ_p for microwave ovens, cordless phones, and Bluetooth devices is 8 ms, 1.25 ms, and 625 μs respectively). In our experiments, using a threshold of $\Delta_{cs} > 0.8$ was able to correctly identify deferring WiFi transmitters (§3.4.2).

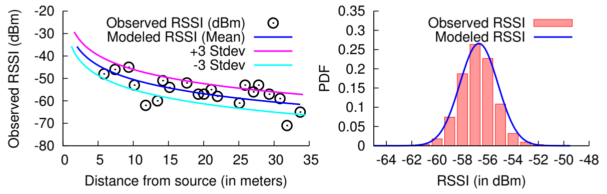


Figure 6: (left) Path loss model created by a WiFiNet APs using WiFi transmissions (right) PDF of actual RSSIs observed at a sample location and the model created using a normal distribution.

Extensions to handle WiFi interference. In general, WiFi links can also experience interference from other WiFi links. We extend our basic approach to measure the overlaps between frame transmissions on a particular WiFi link and the frame transmissions on other WiFi links to compute the probability of frame loss due to hidden interference [21]. In §3.1.4, we experiment with non-WiFi interferers operating alongside hidden terminals and show that WiFiNet is correctly able to identify the true interferer.

2.4 Localizing a non-WiFi device instance

WiFiNet uses a computationally efficient, real-time localization scheme that imposes *zero* profiling overhead, and physically locates the non-WiFi device instance of *unknown transmit power* using a modeling based approach. Below, we explain our localization models.

2.4.1 Model-based localization

Let $\hat{\mathbf{r}} = [\hat{r}_0, \ldots, \hat{r}_{N-1}]$ be the mean RSS vector of all the pulses present in the cluster assigned to a non-WiFi device instance. For localization, we only consider the APs with valid received powers (*i.e.*, $\hat{r}_i \neq \phi$). We divide the entire region into grids of size 0.25×0.25 meters. Let i denote the grid location of AP_i. Let d_{ij} denote the distance between grids i and j. Let $P(l|\hat{\mathbf{r}})$ denote the probability of the non-WiFi device being at location l, given that the received power vector is $\hat{\mathbf{r}}$. We wish to determine the grid location l such that $P(l|\hat{\mathbf{r}})$ is maximized *i.e.*, we want $\text{argmax}_l P(l|\hat{\mathbf{r}})$. Using Bayes' theorem, $P(l|\hat{\mathbf{r}})$ can be written as $P(\hat{\mathbf{r}}|l).P(l)/P(\hat{\mathbf{r}})$. Assuming all locations are equi-probable, and since $P(\hat{\mathbf{r}})$ is constant for all l, we have $\text{argmax}_l P(l|\hat{\mathbf{r}}) = \text{argmax}_l P(\hat{\mathbf{r}}|l)$, which can be calculated as $\text{argmax}_l \prod_{i=1}^{N-1} P(\hat{r}_i|l)$ (assuming independence [23]). Put another way, the grid location l where the non-WiFi device is most likely present can be computed using,

$$\text{argmax}_l \sum_{i=1}^{N-1} \text{log} P(\hat{r}_i|l) \qquad (2)$$

Case of known transmit power (Model-TP). If the non-WiFi device instance is at a grid l, then the *expected* received power at AP_i (located at grid i) can be modeled as a normal distribution $\mathcal{N}(\mu_{il}, \sigma^2)$, where σ is the shadowing variable, and μ_{il} is the *expected mean* of the received power that can be modeled as $\mu_{il} = R^o - 10\gamma log_{10} d_{il}$.

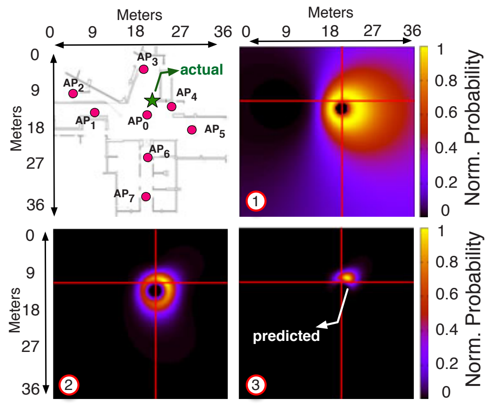


Figure 7: (top, left) Deployment 1 comprising 8 APs. (rest of the sub-figures) FHSS cordless phone device is placed at the starred location. Grid probabilities for predicted phone locations after processing 1, 6 and all AP pairs when using Model-UTP algorithm.

Here, γ is the pathloss exponent and R^o is the power received from the non-WiFi device when placed at a distance of 1 meter from an AP (referred to as *transmit power*). How can we estimate γ for the non-WiFi device? WiFiNet APs derive γ using *WiFi frames i.e.*, each WiFiNet AP uses the data packets or beacons transmitted by neighboring WiFiNet APs to model the propagation loss characteristics (Figure 6) — since both WiFi devices and non-WiFi devices operate on the frequency, the propagation loss characteristics of their transmissions are similar. WiFiNet APs also compute σ^2, by measuring the variance in the received power values (Figure 6 (right)). Knowing μ_{il}, σ and $\hat{r_i}$, the controller can compute $P(\hat{r_i}|l)$ using $\mathcal{N}(\mu_{il}, \sigma^2)$. Intuitively, each AP_i propagates a probability that is maximum around a circle with center at grid i and radius equal to μ_{il}. If the transmit power R^o of the device is known, plugging in $P(\hat{r_i}|l)$ in Equation 2 and iterating over all the grids and APs, we can compute the grid l with the maximum probability of finding the device.

Case of unknown transmit power (Model-UTP). If R^o is not known, we can factor it out by considering each pair of APs: if the non-WiFi device is at a grid l, the *expected difference* in the mean received powers at AP_i and AP_j can be modeled as $\lambda(i, j, l) = \mu_{il} - \mu_{jl} = 10\gamma log_{10}(d_{jl}/d_{il})$, and expected difference in the powers follows a normal distribution with twice the variance: $\mathcal{N}(\lambda(i, j, l), 2\sigma^2)$ [6]. Now, knowing $(\hat{r_i} - \hat{r_j})$, we can compute $P((\hat{r_i} - \hat{r_j})|l)$ and we can localize the non-WiFi device by finding

$$\text{argmax}_l \sum_{i,j} \text{log}P\big((\hat{r_i} - \hat{r_j})|l\big) \quad (3)$$

i.e., each AP pair propagates a probability $P((\hat{r_i} - \hat{r_j})|l)$ on every grid l. The probabilities are high for the grids where the difference in received powers $(\hat{r_i} - \hat{r_j})$ is close to $(\mu_{il} - \mu_{jl})$. After processing all AP pairs, the algorithm outputs the grid l with the maximum probability.

— *Example.* Figure 7 (top, left) shows a deployment of 8 APs along with the location of an FHSS cordless phone (shown using a star). The rest of the figures show how the grid probabilities indicating the location of the phone change after processing 1, 6 and all possible AP pairs.

2.4.2 Alternative localization methods

We also implemented several other localization schemes ranging from simple methods such as (i) *Strongest-AP*, picking the AP with the strongest received power as the device's location, and (ii) *Centroid*, picking the centroid of three APs with the strongest received powers, to more sophisticated approaches like (iii) an *Iterative* approach that performs an exhaustive search over all parameters (γ, σ, R^o, l) to find the grid l with the maximum probability, and (iv) a *Fingerprinting* approach where we collect fingerprints (RSS vectors) at sample locations and localize the device using an approach similar to [5]. In §3, we compare our model based approaches to all these methods.

3 Experimental Results

We break our evaluation into four parts: (i) we validate WiFiNet's non-WiFi device interference estimates across a variety of scenarios, (ii) we evaluate WiFiNet's accuracy in physically locating the non-WiFi devices, (iii) we emulate a non-WiFi interference prone enterprise WLAN scenario and show WiFiNet's utility in such a setting and (iv) we benchmark different components of WiFiNet and highlight cases where WiFiNet's performance could degrade. We start by presenting the details of our implementation.

Implementation. We implemented WiFiNet using commodity WiFi APs equipped with Atheros AR9280 wireless cards that are connected to a central controller over the Ethernet. Our implementation consists of few hundred lines of C code and 9800 lines of Python scripts that implement non-WiFi device detection functionality at the APs [16], perform synchronization across multiple APs, and implement clustering algorithms, interference analysis and device localization methods at the controller.

Evaluation set up. We experiment with devices in 2.4 GHz spectrum, and our current prototype has been tested with 5 different non-WiFi devices types : (i) high duty devices (analog cordless phones), (ii) fixed-frequency pulsed transmitters (ZigBee devices), (iii, iv) two types of frequency hopping devices (FHSS cordless phones, Bluetooth devices), and (v) broadband interferers (microwave ovens). We run our experiments on two different deployments: (i) Deployment 1 used 8 APs (Figure 7) and (ii) Deployment 2 used 4 APs (Figure 17). We experiment with different non-WiFi device locations, 802.11 rates, channel conditions and traffic patterns: (i) UDP with saturated traffic as well as reduced traffic loads, and (ii) replay of real HTTP/TCP wireless traces (§3.1.5). Unless otherwise stated, we run WiFi links on 802.11 rate to 6 Mbps and use backlogged UDP traffic with a packet size of 1400B.

Ground truth. We use an approach similar to bandwidth tests [21, 11] (conventionally used to determine interference between WiFi links) to determine the ground truth impact of a non-WiFi interferer on a WiFi link: we perform controlled experiments using backlogged traffic on the WiFi link and we (i) measure $p[L]$, the loss rate when the interferer is inactive, and (ii) measure $p[I \cup L]$, the loss rate when the interferer is active. We can then measure the ground truth *i.e.*, the actual $p[I|O]$ (§2.3) [1]. For experiments involving multiple devices, we measure the ground truth by activating only one device at a time and measuring its impact on the link. We note that the same ground truth ($p[I|O]$) is valid when multiple devices are activated simultaneously (the overall impact $p[I]$ may change, but $p[I|O]$ remains the same). WiFiNet, however, computes $p[I|O]$ estimates in presence of multiple, simultaneously active devices and WiFi links using any traffic load.

Metrics used. For interference estimation, we compare WiFiNet's real-time, passive interference estimate of "impact given overlap" ($p[I|O]$) with that obtained using controlled experiments wherein the device is activated in isolation (ground truth). For localization, we report the difference in the actual and the predicted location of the non-WiFi device (*i.e.*, localization error) in meters.

3.1 Validating Interference Estimates

We start by validating WiFiNet's interference estimates across a variety of scenarios.

3.1.1 Single interferer scenarios

Method. We experiment with a total of 165 link-interferer scenarios comprising 4 non-WiFi devices — a microwave oven, an analog cordless phone, an FHSS cordless phone and a ZigBee transmitter. We activate each device in turn, and place it at different distances to vary the interference on the monitored WiFi link. We compute the ground truth (actual $p[I|O]$) using controlled experiments that measure the link loss rate when the device is active and that when the device is inactive. Next, we *randomly* activate and de-activate the non-WiFi device while the WiFi link is active and measure WiFiNet's real-time estimate.

Results. Figure 8 (left) shows that WiFiNet correctly estimates a non-WiFi device's impact — across all device types and different amounts of interference (ranging from weak to strong), WiFiNet's estimates lie close to the ground truth (the points lie close to $y = x$). Figure 8 (right) shows that the overall error in WiFiNet's estimate

[1]Knowing $p[O]$, it is possible to determine $p[I|O]$. Details of the derivation are presented in our technical report [19].

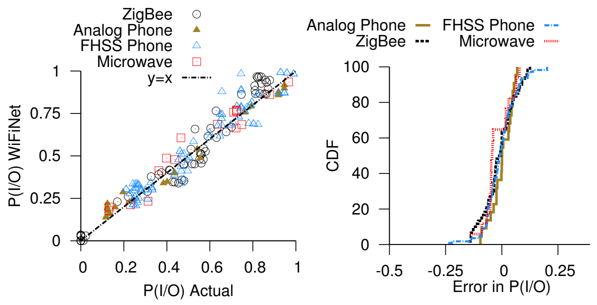


Figure 8: (left) Interference estimates obtained using controlled measurements (ground truth) and WiFiNet on 165 link-interferer scenarios comprising 4 different classes of devices. (right) CDF of error in interferer estimates is within ± 0.1 for 95% of the cases.

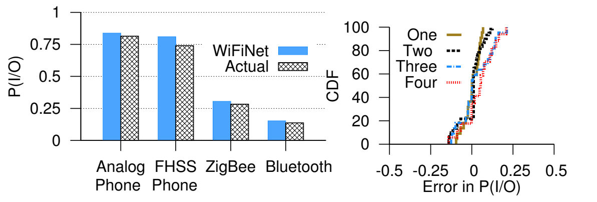


Figure 9: Accurately identifying impact of each interferer in the presence of multiple non-WiFi devices. (left) example scenario showing WiFiNet is able to identify the strong interferers (analog cordless phone, FHSS phone) and weak interferers (ZigBee and Bluetooth devices) accurately. (right) CDF of error in inteference estimates in the presence of multiple interferers.

is within ± 0.1 for more than 95% of the cases for all 4 devices.

3.1.2 Multiple interferers of different types

Method. In each run, we choose upto 4 random devices of different types, place them at random locations, randomly activate and de-activate them, creating scenarios when these devices are *simultaneously* active and measure WiFiNet's interference estimate for each device. For ground truth, we activate only one device at a time and perform controlled measurements. We repeat the experiments for different combinations of devices and locations.

Results. Figure 9 (left) shows a particular run which comprised two strong interferers (analog phone and FHSS cordless phone) and two weak interferers (ZigBee and Bluetooth devices). We find that WiFiNet is not only able to accurately identify the strong and weak inteferers, but is also able to discern the exact impact of each of these devices in spite of them being active simultaneously. Figure 9 (right) shows the CDF of error in interference estimates for combinations of 2, 3 and 4 devices across 60 runs. While the overall error slightly increases with increase in the number of devices, the error is within ± 0.15 for more than 85% of the cases even when operating 4 devices. The slight increase in error is due to increased overlap in the transmissions from multiple devices. We benchmark the effect of overlapping transmissions in §3.4.4.

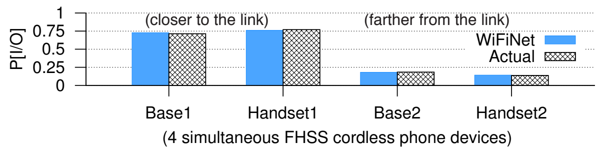


Figure 10: WiFiNet**'s accuracy in the presence of multiple non-WiFi devices of the same type. Out of** 4 **FHSS cordless phone devices,** 2 **are placed close to the link, and** 2 **are placed farther away.**

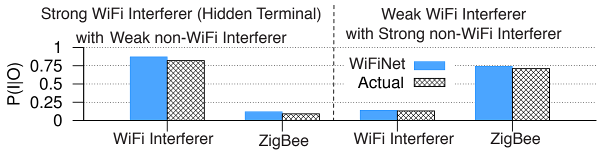


Figure 11: Estimating the interference impact of a WiFi interferer (hidden terminal) and a non-WiFi interferer (ZigBee device).

3.1.3 Multiple interferers of the same type

Method. We now evaluate WiFiNet's performance when simultaneously operating multiple devices of the *same type*. We use 4 FHSS cordless phone devices — one base/handset pair is placed close to the WiFi link (to create strong interference), whereas the other pair is placed farther away (to create weak interference).

Results. Figure 10 shows that WiFiNet is able to (i) accurately identify all 4 FHSS cordless phone devices using clustering mechanisms (benchmarked in §3.4.3) and (ii) accurately identify strong interferers (base/handset pair placed close to the link) and weak interferers (base/handset pair placed farther away from the link).

3.1.4 Mix of WiFi and non-WiFi interference

Method. We evaluate WiFiNet's accuracy when simultaneously operating a WiFi interferer (hidden terminal) and a non-WiFi interferer (ZigBee device). The interferers are placed at different distances from the monitored WiFi link to create two scenarios: (i) strong WiFi interferer with a weak non-WiFi interferer (ii) weak WiFi interferer with a strong non-WiFi interferer. The WiFi interferer's traffic follows an http on-off model for with sleep and active times derived from a wireless trace [17], whereas the ZigBee device used a constant bit rate. As before, to measure ground truth, we operate the devices in isolation.

Results. Figure 11 shows the results. In case (i), WiFiNet finds that losses are more likely to happen when the monitored link's frames overlap with WiFi interferer's frames, whereas in case (ii), the losses show a high correlation when frames overlap with non-WiFi device's transmissions, resulting in accurate estimates for both cases.

3.1.5 Dynamic interference settings

Handling WiFi client mobility. We now evaluate WiFiNet's ability in updating the interference estimates that reflect the changing impact of a non-WiFi interferer due to client mobility. We use the set up shown in Figure 12 (top) where in a WiFi client is moving away

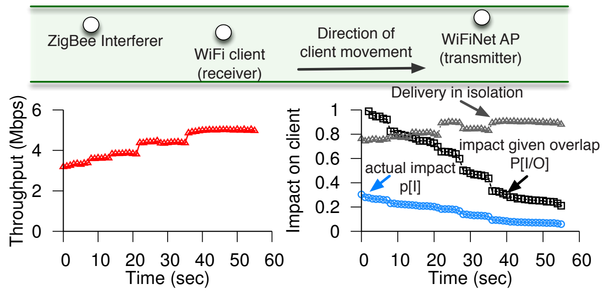


Figure 12: WiFiNet**'s ability to track the changing interference patterns for a client that is moving away from a ZigBee interferer. (left) instantaneous throughput at the client (right) delivery in isolation (*i.e.*, in absence of overlap), impact given overlap ($p[I|O]$) and actual impact ($p[I]$) are shown.**

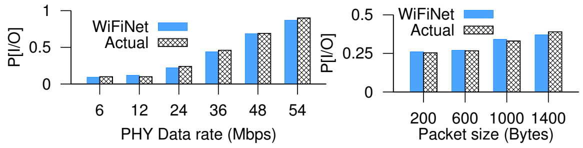


Figure 13: Impact of (i) PHY rate and (ii) packet size on $p[I|O]$ in presence of a ZigBee interferer. For (i), packet size is fixed at 1400 bytes, and for (ii), rate is fixed at 12 Mbps. $p[I|O]$ rises sharply with rate, the change in $p[I|O]$ with packet size is less pronounced.

from a ZigBee interferer. In the figure, plot on the left shows the instantaneous throughput at the client increases as it moves away from the interferer. The plot on the right shows WiFiNet's ability to track (i) delivery in isolation (*i.e.*, in the absence of overlap) that shows a slight increase, (ii) the impact given overlap $p[I|O]$, which rapidly drops down from 0.98 to 0.2 as the client moves farther away and (iii) the actual impact $p[I]$, owing to the probability of overlap, drops from 0.3 to 0.12. The decrease in the actual impact closely matches with the increase in throughput confirming WiFiNet's utility in understanding client performance in dynamic wireless environments.

Variable 802.11 rates and packet sizes. We evaluate WiFiNet's ability to dynamically track the changing interference estimates due to changes in (i) PHY rates and (ii) packet sizes used by the links. For ground truth, we perform controlled experiments at each PHY rate, whereas for WiFiNet we enable dynamic rate adaptation using SampleRate and capture the estimates in real-time. Figure 13 (left) shows that WiFiNet's estimates derived from rate adaptation closely match the ground truth. Since higher rates require higher SINR to decode a frame successfully, impact of the interferer *increases* with the increase in rate. Next, we fix the PHY rate (to 12 Mbps) and repeat our experiments for different packet sizes. Figure 13 (right) shows that WiFiNet is correctly able to track the slight increase in the interferer's impact at larger packet sizes.

Replay of wireless traces. We evaluate WiFiNet's performance using publicly available Sigcomm 2004 traffic traces [17]. We partitioned the trace into heavy, medium,

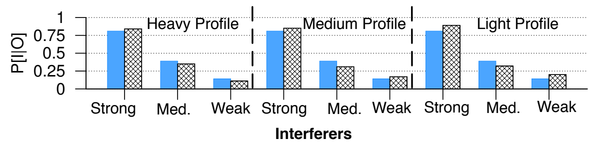


Figure 14: WiFi links replay real HTTP/TCP wireless traces [17] (heavy, medium, and light profiles) in presence of strong, medium and weak interferers. WiFiNet's **estimates closely match the ground truth in each case. The slight mismatch is due to the variability in packet sizes as the ground truth was measured using** 1400 **byte packets, whereas the traces comprised packets of different sizes.**

Delay	Min.	25th %ile.	median	75th %ile	Max.
Convergence time	319 ms	549 ms	972 ms	1.7 sec	3.6 sec

Table 1: Distribution of convergence time for WiFi links replaying HTTP/TCP wireless traces (heavy, medium and light profiles) in presence of an FHSS cordless phone interferer.

and light periods corresponding to periods with airtime utilization of more than 50%, between $20-50\%$, and less than 20% respectively, at different times of the conference [21]. The HTTP/TCP sessions are then replayed on WiFi links (using the mechanism described in [7]) in the presence of strong, medium and weak ZigBee interferers. Each client emulated the behavior of one real client from the trace, faithfully imitating its HTTP transactions. Figure 14 shows that that WiFiNet's interference estimates are close to that of the ground truth across different traffic profiles and interfering scenarios. The slight differences between the estimates are due to the variability in packet sizes in the real traces, compared to the ground truth that was measured using 1400 byte packets. We also show the CDF of time taken by WiFiNet to converge to the right $p[I|O]$ estimates in Table 1 (median $<$ 1 sec). We benchmarks the factors affecting convergence time in §3.4.1.

3.2 Accuracy of Localization

We now evaluate our localization algorithms.

3.2.1 Accuracy across different classes of devices

Figure 15 shows CDF of localization error for two non-WiFi device types: (i) frequency-hopping cordless phone and (ii) high duty, analog cordless phone, when using deployment 1 with 8 APs shown in Figure 7. As shown in the figure, the floor's dimensions were 36 meters * 36 meters. Devices were placed at random locations and for each location, we compute the difference in the predicted and actual location for 5 different localization schemes (§2.4). We find that all algorithms perform well, resulting in a median error of $1-3$ meters for the FHSS phone, and $1.7-4$ meters for the analog phone. Here, WiFiNet's modeling based localization approaches perform similar to the Iterative approach that employs an exhaustive search, and is better than Fingerprinting (§2.4.2) that incurs a profiling overhead. Accuracy of Fingerprinting, however, can be improved by increasing the density of fingerprints (0.05/sq.meter in this case) at the cost of a higher profiling overhead.

Algorithm	Min. error	25th %ile.	median	75th %ile	Max error
Iterative	0.3m	0.8m	2.1m	4m	10m
Model-TP	0.3m	0.3m	1.3m	3m	8m
Model-UTP	0.3m	0.8m	1.3m	4m	11m

Table 2: Overall localization error for an analog cordless phone and an FHSS phone when placed at random locations in deployment 2.

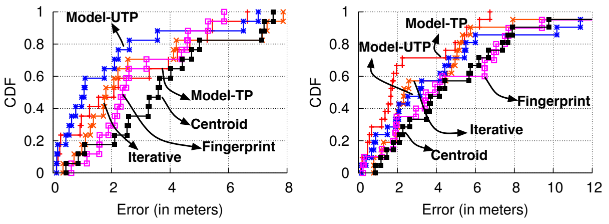


Figure 15: Accuracy of localization for (left) FHSS cordless phone and (right) analog cordless phone for deployment 1 (Figure. 7).

3.2.2 Effect of AP density

In each run, we randomly chose a subset of 4 APs (out of the 8 APs in deployment 1) and compute the localization error. We repeat the experiment for 25 runs and report the average error in Figure 16 (left). We observe that when the density of the AP deployment is sparse, the performance of Centroid algorithm worsens (median error of 8 meters) compared to the other algorithms (median error of 2.5 to 4.8 meters). Figure 16 (right) shows the degradation in the performance of Model-UTP, when the number of APs is reduced from 8 to 3. The median error only increases from 1 meter to 4 meters indicating the better performance of modeling based approaches in sparse deployments.

3.2.3 Improvements with fine-grained modeling

To understand the benefits from using a per-AP path loss exponent, we compare the performance of our modeling-based localization approaches when a uniform path loss exponent is used. Table 3 shows that when switching to a uniform path-loss exponent, the median error increased from 1.7 to 3.6 meters, and the maximum error increased from 6.7 meters to 12 meters. Using a per-AP path loss improves the WiFiNet's localization accuracy as it takes into account the differences in the environments surrounding the APs (e.g., walls and other obstacles).

3.2.4 Location insensitivity

We repeated our experiments to benchmark the performance of our algorithms in a different topology and environment (deployment 2 with 4 APs, Figure 17). Table 2 shows the overall error for the modeling-based and Iterative approaches. We find that the algorithms perform well with a median error of $1.3-2.1$ meters.

3.3 Emulating an Enterprise WLAN

We now try to emulate the structure of our in-building WLAN by placing a WiFiNet AP near each production AP and distribute clients into offices (Figure 17). Our

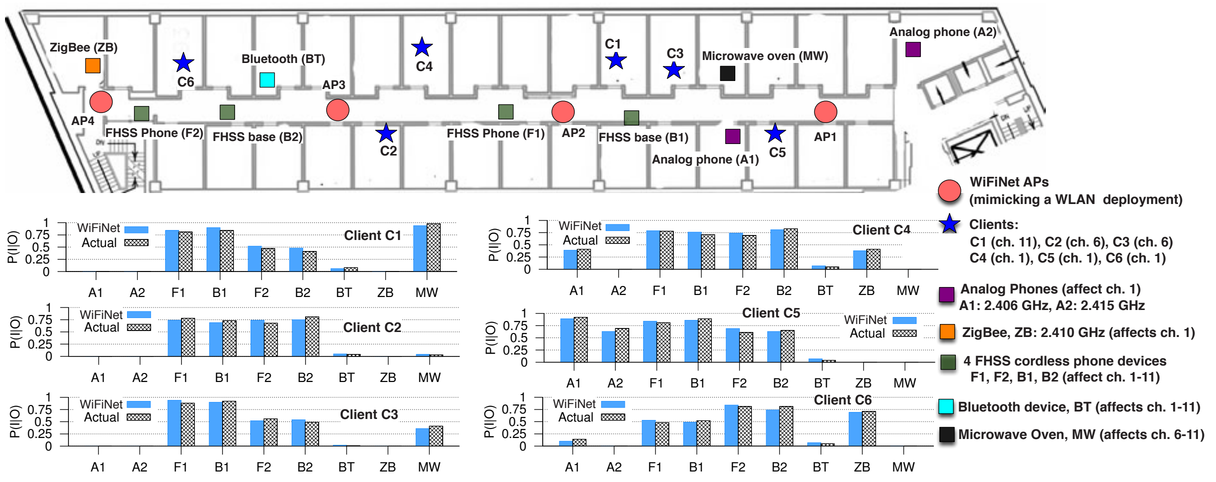


Figure 17: (Deployment 2) Emulating an enterprise WLAN with 4 APs and 6 clients. A total of 9 non-WiFi devices are placed to interfere with the clients: 2 analog phones, 4 FHSS cordless phone devices, a Bluetooth device, a ZigBee device and a microwave oven. WiFiNet **is able to accurately characterize the interference impact ($p[I|O]$) of all devices (even those of the same type) on each of the clients.**

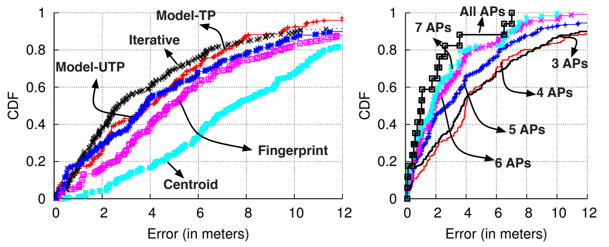


Figure 16: Localization accuracy for FHSS cordless phone (left) for subsets of 4 APs from deployment 1 (right) using Model-UTP when the number of APs was decreased from 8 to 3.

Scheme	Min. error	25th %ile.	median	75th %ile	Max error
Uniform γ	0.2m	1.9m	3.6m	7m	12m
per-AP γ	0.2m	2.0m	1.7m	2.3m	6.7m

Table 3: Overall localization error for the Model-TP algorithm with (i) uniform and (ii) per-AP path loss exponents (deployment 1).

topology consists of 4 APs and 6 clients. We use a total of 9 non-WiFi interferers: 2 analog phones (high duty devices), 4 FHSS cordless phone devices, a Bluetooth device (frequency hopping devices), a ZigBee device (fixed frequency, pulsed transmitter) and a microwave oven (broadband interferer). WiFi links are assigned channels (shown in Figure 17) so as to create a scenario where each non-WiFi device affects at least one link. Each WiFi link follows an HTTP traffic model, with on-off times derived from a wireless trace [17]. We activate and deactivate the non-WiFi devices randomly, creating scenarios when devices are simultaneously active. As before, for ground truth measurements, we activate only one device at a time.

Figure 17 shows the interference impact of each interferer on the WiFi links — depending on the channel of operation, location of the client, and overlap probability (based on the actual WiFi traffic and non-WiFi device activity), WiFi links experience different amount of interference from each non-WiFi device. Further, WiFiNet's estimate *closely matches* the ground truth for each case. We find that all 4 FHSS cordless phone devices affect *all* the WiFi links ($p[I|O]$ varied from 0.45 to 0.8 due to their high transmit power of -20 dBm). The overall impact $p[I]$, however, only varied from 0.1 to 0.31 owing to their frequency hopping nature. Peak emissions of microwave ovens are typically in 2.45 to 2.47 GHz, and so the oven severely affected the client $C1$ which operated on channel 11. It is interesting to note that $C3$ (operating on channel 6) was also affected by the oven ($p[I|O]{=}0.36$) as it was close to the device, whereas $C2$ (channel 6, farther from the device) and $C5$ (channel 1, closer to the device) were not affected. Bluetooth device, due to its low power and adaptive frequency hopping nature did not significantly affect any of the links. On the other hand, high powered and high duty device like analog phones (A1 and A2) affected the clients on channel 1 (C4, C5, C6) much more than the ZigBee device that had a lower transmit power.

3.4 Microbenchmarks and Other results

We now benchmark convergence time, clustering algorithms, highlight WiFiNet's limitations and present results on estimating sender-side interference.

3.4.1 Convergence time

We define the convergence time as the time taken by WiFiNet to gather sufficient samples (*i.e.*, overlaps between WiFi frames and non-WiFi transmissions) to compute an accurate $p[I|O]$ estimate (within ± 0.1 of the ground truth). Figure 18 (left) shows 9 different scenarios where a ZigBee interferer causing strong, medium or weak interference is activated along with a WiFi link. Across all scenarios, we find that < 100 overlaps between WiFi frames and ZigBee transmissions are enough for $p[I|O]$ to converge (convergence points shown with black circles). Across different non-WiFi interferers and links

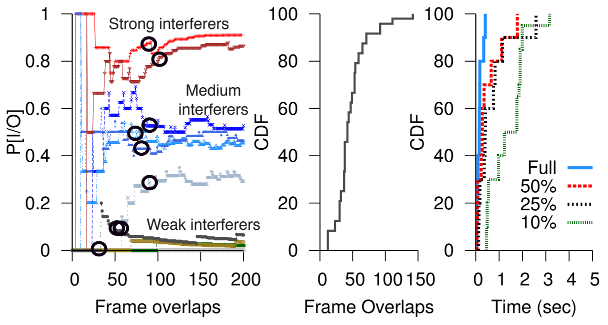


Figure 18: (left) Number of frame overlaps are required to converge for 10 ZigBee interferer scenarios including strong, medium and weak interference (middle) CDF of the number packet overlaps required for $p[I|O]$ to converge (right) Convergence time as a function of the traffic load for an FHSS cordless phone.

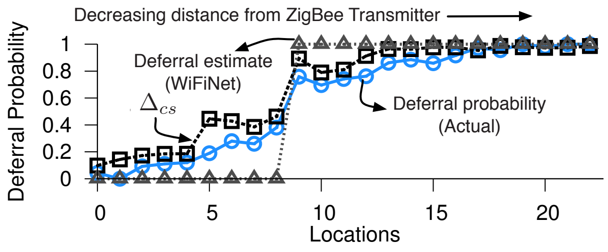


Figure 19: WiFiNet's estimates of deferral probability close match the ground truth. Here, a WiFi transmitter is moving toward a ZigBee interferer leading to increase in the deferral probability.

$<$ 150 overlaps are enough to converge to the ground truth (CDF shown in Figure 18 (middle)). The time for convergence depends on the WiFi link's traffic load, and the activity of the non-WiFi device. Figure 18 (right) shows that although the convergence time increases with lesser traffic, it is less than 4 seconds across a variety of traffic loads when using an FHSS cordless phone as an interferer. For devices like microwave ovens and analog cordless phones, convergence time was much lesser owing to increased overlaps.

3.4.2 Estimating sender-side interference

We also benchmarked WiFiNet's ability to correctly estimate the carrier sensing interference across a number of non-WiFi devices and links. Due to lack of space, we only show one result in Figure 19. Here, we move a WiFi transmitter toward a ZigBee device (periodically transmits 4 ms pulses) and measure its deferral probability (§2.3). For ground truth, we measure the transmitter's sending rate when the device is active and that when the device is inactive. WiFiNet estimates the deferral probability in real-time — we observe that Δ_{cs} *i.e.*, the fraction of Case (2) instances (§2.3) increases as we move the transmitter away, indicating increased deferral. Further, Δ_{cs} also closely matches the ground truth.

3.4.3 Performance of clustering

Clustering is straightforward in many cases e.g., when the devices are of different types, or in the case of fixed-frequency devices (of the same type) using different center frequencies. We benchmarked our RSS and timing based clustering algorithms (§2.2) for the harder cases of (i) fixed-frequency devices using the same center frequency and (ii) frequency hopping devices. Table 4 shows the overall summary (when operating up to 4 devices of the same type). We find that clustering algorithms perform reasonably well with $> 88\%$ accuracy in detecting the number of device instances. In case of over-clustering, the number of pulses in the extra clusters were relatively low, allowing us to discard the false positives. Under-clustering, however, can lead to error in estimates that can happen if the devices are close to each other (§3.4.4). Using timing attributes (when available) results in increased accuracy, compared to RSS based clustering, as timing attributes are not sensitive to the distance between devices (§3.4.4). Also, k-means+EM clustering has higher accuracy compared to density based clustering (DBSCAN).

Algorithm	Attribute	Clustering performance % Correct	% Over-cluster	% Under-cluster
DBSCAN	Timing	92.7%	5%	2.3%
DBSCAN	RSS	88.7%	5.2%	6.1%
k-Means + EM	Timing	97.6%	1.3%	1.1%
k-Means + EM	RSS	91.4%	6.5%	2.1%

Table 4: Performance of clustering mechanisms used in WiFiNet. Results for two clustering algorithms (DBSCAN and k-means+EM) using (i) start time offset and (ii) RSS attributes are shown. Up to non-WiFi devices of the same type were placed at random locations.

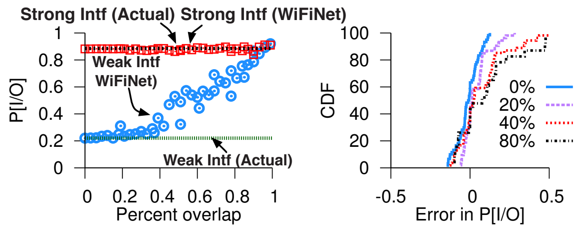


Figure 20: (left) Ability of WiFiNet to correctly identify interferers when transmissions from two non-WiFi devices overlap. $p[I|O]$ measured by WiFiNet for both strong ($p[I|O] = 0.88$) and weak ($p[I|O] = 0.22$) interferers as a function of their overlap in transmission times. If the overlap is less than 45%, WiFiNet can distinguish the strong and weak interferers accurately. (right) Ability of WiFiNet to correctly estimate $p[I|O]$ of an interferer as function of percentage of pulses lost (*i.e.*, not captured) by an WiFiNet AP.

3.4.4 Sources of error

We now highlight some of the scenarios where WiFiNet's performance can degrade.

Overlapping transmissions. We now benchmark the effect of transmission overlaps between multiple interferers. Figure 20 (left) shows WiFiNet's interference estimates in the presence of a strong and a weak non-WiFi interferer, as a function of the overlap between their transmission times. In the unlikely case when the transmissions from both non-WiFi devices overlap 100% of the time, WiFiNet is unable to distinguish between the two. However, as the percentage of overlap decreases, WiFiNet is able to discern the impact of the weak interferer. In practice, we expect diversity in device transmission times [16] to allow WiFiNet to output accurate interference estimates.

Coverage. WiFiNet's ability to derive an accurate interference estimate depends on how well the non-WiFi device's transmission are captured. In particular, $p[I|O]$ and $p[L]$ estimates will differ from the ground truth when none of the WiFiNet APs capture the device's transmissions. Figure 20 (right) shows the impact of losing non-WiFi transmissions — the error in estimates increase with decrease in the percentage of captured transmissions. In a typical enterprise deployment with multiple APs, this might not be a concern as we can expect at least one AP to capture the device's transmissions.

4 Related Work

Device detection and interference estimation. Commercial solutions such as Wispy [4], Cisco Spectrum Expert [2] and Bandspeed AirMaestro [1] use custom hardware (signal analyzer ICs) to detect RF devices operating in the medium. However, these solutions do not provide the capability to estimate the interference caused by the non-WiFi devices to the the WiFi links. Recent research work such as DOF [8], RFDump [13], TIMO [18] can also detect the presence of non-WiFi device activity using specialized hardware such as channel sounders and software-defined radios. Such platforms enable TIMO and DOF to go beyond detection and employ signal processing techniques to mitigate interference and develop mechanisms to co-exist with non-WiFi devices. WiFiNet takes a step towards empowering APs and clients with such functionality, by providing non-WiFi interference estimation capability under the constraints of commodity WiFi hardware. In [14], the authors use a single WiFi card to infer interference from Bluetooth and microwave ovens by analyzing the timing of WiFi packet errors. However, their technique does not generalize to detect inteference other non-WiFi devices that don't exhibit timing properties (e.g., ZigBee) and cannot distinguish between devices of same type. In comparison, WiFiNet can also estimate the interference from multiple, simultaneously operating devices and pin-point their location in the physical space.

Device localization. There has been limited prior work on designing a generic system to localize the various non-WiFi devices on the top of commodity WiFi hardware. Existing literature has looked at localizing specific device types (e.g., Bluetooth [20], Zigbee [9]) by using sensors of the same type. Amongst commercial solutions, Wi-Spy device finder [4] uses a directional antenna and requires a user to walk and manually search for the location of the transmitter. Cisco CleanAir [2] finds the location of RF transmitter sources by using specialized hardware in the access points. WiFiNet uses only commodity WiFi cards to not only detect the location of non-WiFi devices, but also estimate their interference impact.

5 Conclusion

We presented WiFiNet, a system to estimate the interference experienced by WiFi links in presence of non-WiFi devices using only WiFi hardware. WiFiNet can correctly estimate the impact of each non-WiFi device, in presence of multiple other interferers, even if they are of the same type. It also correctly tracks changes due to client mobility, dynamic traffic loads, and varying channel conditions. Further, WiFiNet also identifies the physical locations of non-WiFi devices. We believe a system such as WiFiNet can help WLAN administrators use commodity WiFi APs to better understand and manage non-WiFi interference, especially in enterprise WLANs.

Acknowledgments: We would like to thank the anonymous reviewers and our shepherd Kate Lin, whose comments helped bring the paper into its final form. All authors are supported in part by the following grants of the US NSF: CNS-1040648, CNS-0916955, CNS-0855201, CNS-0747177, CNS-1064944, and CNS-1059306.

References

[1] Bandspeed AirMaestro. http://www.bandspeed.com/.
[2] Cisco Spectrum Expert. http://tinyurl.com/5eja7f.
[3] The WEKA data mining software: an update. ACM SIGKDD Explorations Newsletter.
[4] Wi-Spy spectrum analyzer and device finder. www.metageek.net.
[5] P. Bahl and V. N. Padmanabhan. Radar: an in-building rf-based user location and tracking system. In *Infocom'00*.
[6] B.Lin and J. Wu. Analysis of hyperbolic and circular positioning algorithms using stationary signal-strength-difference measurements in wireless communications. In *VTC*, 2003.
[7] J. Eriksson, S. Agarwal, P. Bahl, and J. Padhye. Feasibility study of mesh networks for all-wireless offices. In *MobiSys*, 2006.
[8] S. Hong and S. Katti. DOF: A local wireless information plane. In *ACM SIGCOMM 2011*.
[9] J. Blumenthal et. al. Weighted centroid localization in zigbee-based sensor networks. *IEEE ISISP'07*.
[10] J. Han and M. Kamber. Data Mining: Concepts and Techniques.
[11] J. Padhye et al. Estimation of link interference in static multi-hop wireless networks. In *IMC'05*.
[12] J. Sander et. al. Density-based clustering in spatial databases. *Data Mining Knowledge Discovery'98*.
[13] K. Lakshminarayanan et. al. RFDump: an architecture for monitoring the wireless ether. In *CoNext'09*.
[14] K. Lakshminarayanan et. al. Understanding 802.11 performance in heterogeneous environments. HomeNets '11.
[15] R. Mahajan et. al. Analyzing the mac-level behavior of wireless networks in the wild. *SIGCOMM '06*.
[16] S. Rayanchu, A. Patro, and S. Banerjee. Airshark: Detecting non-WiFi devices using commodity WiFi hardware. In *ACM IMC'11*.
[17] M. Rodrig, C. Reis, R. Mahajan, D. Wetherall, J. Zahorjan, and E. Lazowska. CRAWDAD data set uw/sigcomm2004.
[18] S. Gollakota et al. Clearing the RF Smog: Making 802.11 Robust to Cross-Technology Interference. In *ACM SIGCOMM 2011*.
[19] S. Rayanchu et. al. Catching whales and minnows using wifinet: Deconstructing non-wifi interference using wifi hardware. Computer sciences, UW Madison Technical Report TR1712.
[20] S. Thongthammachart and H. Olesen. Bluetooth enables in-door mobile location services. *VTC'03*.
[21] V. Shrivastava et. al. PIE in the Sky: online passive interference estimation for enterprise wlans. In *NSDI'11*.
[22] Y. Cheng et. al. Jigsaw: Solving the puzzle of enterprise 802.11 analysis. In *SIGCOMM'06*.
[23] M. Youssef and A. Agrawala. The horus wlan location determination system. In *MobiSys'05*.

RPT: Re-architecting Loss Protection for Content-Aware Networks

Dongsu Han, Ashok Anand†, Aditya Akella†, Srinivasan Seshan
Carnegie Mellon University †*University of Wisconsin-Madison*

Abstract

We revisit the design of redundancy-based loss protection schemes in light of recent advances in content-aware networking. Content-aware networks minimizes the overhead of redundancy, if the redundancy is introduced in a way that the network can understand. With this insight, we propose a new loss protection scheme called redundant packet transmission (RPT). Using redundant video streaming as an example, we show that our approach, unlike FEC in traditional networks, provides low latency with high robustness and is insensitive to parameter selection. We tackle practical issues such as minimizing the impact on other traffic and the network. We show that RPT provides a simple and general mechanism for application-specific control and flow prioritization.

1 Introduction

A variety of current and future Internet applications require time critical or low latency communication. Example applications include delay-sensitive live/interactive video streams, online games, and video-based calls (e.g., Apple's FaceTime), all of which send real-time data. Studies of real-time systems [36, 48] suggest that the maximum tolerable one-way delay is around 150ms for real-time interaction. Within a data center, many soft real-time applications that interact with users require low latency communication [13]. Certain classes of inter-datacenter transfers, such as mirroring financial data, also require real-time communication [17].

The central challenge in supporting such delay-sensitive real-time applications is protecting them from network loss. One set of conventional approaches—acknowledgment-based retransmission protocols—are not appropriate for real-time communication as retransmissions triggered by timeouts can take several RTTs and violate applications' timing constraints [44, 51]. Another set of approaches—redundancy-based schemes such as Forward Error Correction (FEC)—suffer from a fundamental tension between robustness and the bandwidth overhead [20, 25], making them either difficult to tune or inefficient in practice.

These techniques have been tuned to provide the best performance tradeoffs possible in traditional networks. In contrast, the focus of our paper is to show that better protection against congestion losses may be possible in *content-aware networks*. We use the term content-aware networks to refer to the variety of architectural proposals [14, 32, 37, 42] and devices [2, 4, 10, 35] that cache data and remove duplicates to alleviate congestion (i.e., they perform *content-aware* processing of packets). Content-aware processing is seeing ever-growing adoption in a variety of settings, including mobile and cellular networks [11], data centers [35], cloud computing [9], and enterprise networks [4]. The most popular of such content-aware network devices are the WAN optimizers [5, 8, 10] that are typically placed at branch and main offices or between data-centers to reduce the traffic between them. Market reports indicate that the market for such devices is growing rapidly [4].

Our core assumption is that content-aware network devices will be widely deployed across a variety of links in future networks. Given this setting, we ask: *(i)* How do we re-architect loss protection for delay sensitive applications operating in this new context? *(ii)* Does content-awareness help simplify or further complicate the issues that existing loss protection schemes face? And, why?

We show that taking content-awareness into account challenges the conventional wisdom on the trade-offs of redundancy in protecting against losses in time-critical and delay-sensitive applications. In particular, we show that it is now possible to use redundancy in a simple yet clever fashion to ensure robustness against congestion losses while imposing little or no impact on the network or on other existing applications. Equally importantly, we show that it is now far easier to integrate loss protection with other design constraints such as adhering to tight delay bounds.

We believe that the duplicate suppression actions in content-aware frameworks provide a tremendous opportunity to use redundancy-based protection schemes. However, redundancy must be introduced in the right way to ensure: *(a)* the network can eliminate it optimally to provide the desired efficiency and *(b)* the impact on other applications can be controlled.

We describe Redundant Transmission (RT) – a loss protection scheme that intelligently sends multiple copies of the same data. The basic idea of RT is to expose the redundancy directly to the underlying content-aware network.

The simplest form of RT is to send multiple copies of the same packet. When packets are not lost, the duplicate transmissions in RT are compressed by the underlying network and add little overhead. In contrast, when the network is congested, the loss of a packet prevents the compression of a subsequent transmission. This ensures that the receiver still gets at least one decompressed copy of the original data. In some situations, the fact that packet losses do not directly translate to less bandwidth use may raise the concern that RT streams obtain an unfair share of the network. However, existing congestion control schemes, with some critical adjustments to accommodate RT behavior, can address this concern.

In essence, RT signals the network the relative importance of packet by transmitting multiple copies. RT requires almost no tuning; this stands in stark contrast with the difficulty of fine-tuning FEC-based approaches for traditional networks. Finally, RT decouples redundancy from delay and easily accommodates application timing constraints; in comparison, FEC schemes today closely tie delay guarantees with redundancy encoding since the receiver cannot reconstruct lost packets until the batch is complete. In effect, RT on content-aware networks can effectively support a variety of time-critical applications far better than existing approaches for traditional networks.

To illustrate the benefits of RT concretely, we use as an example RPT, a simple variant of RT for real-time video in a redundancy elimination network. Our evaluation of RPT, using a combination of real-world experiments, network measurements and simulations, shows that RPT decreases the data loss rate by orders of magnitude more than FEC schemes applicable to live communications. As a result, it achieves better video quality than FEC for a given bandwidth budget, or uses up to 20% less bandwidth than FEC schemes to deliver the same video quality.

We make the following contributions in this paper:

1. We highlight the need to reconsider the design of loss protection for content-aware networks. We show that network content-awareness enables vastly simpler and more effective approaches to loss protection (§3).

2. We describe a redundancy scheme, RT, that can provide a high degree of robustness at low overhead, and require minimal tuning (§3 and §4).

3. Through extensive experiments and simulations, we find that RT can improve the robustness of real time media applications with strict timing constraints (§6).

In the remaining sections, we review related work (§2), discuss realistic deployment examples as well as general implementation of RT on other content-aware networks (§5), and finally conclude in §7.

2 Current Loss Recovery Schemes

Packet losses are often inevitable on the Internet, especially across heavily-loaded links, such as cross-country or trans-continental links. Many prior works use *timeout-based retransmission* to recover lost data [18, 27, 44, 49, 51] on traditional networks. However, retransmission causes large delays [51] which are often difficult to hide. Also, the performance depends on correct timeout estimation [44] which is often non-trivial [18, 44]. Because of these intrinsic limitations, more sophisticated enhancements such as selective retransmission [27, 49], play-out buffering [49], and modification to codecs [51] are often required to augment retransmission based loss recovery.

Another option is *redundancy-based recovery*, with FEC being an example framework that is widely used today. While coding provides resilience, the use of FEC is constraining in many ways in practice: *(1)* In FEC, the receiver cannot recover lost packets until the batch is complete. This limits the size of the batch for delay-sensitive applications. For example, at most 5 packets are typically batched in video chat applications such as Skype [56]. *(2)* Small batch size makes FEC more susceptible to bursty loss. For example, adding a single coded FEC packet for every five original data packets is not enough to recover from two consecutive lost packets. Therefore, in practice, the amount of redundancy used is high (e.g., 20% to 50% [20, 25, 56]), which is much higher than the underlying packet loss rate. *(3)* Furthermore, FEC needs to adapt to changing network conditions [19, 57], which makes parameter tuning even more difficult. Many studies [29, 57] have shown that fine tuning FEC parameters within various environments is non-trivial.

More sophisticated redundancy-based recovery schemes such as fountain codes [21], rateless coding with feedback [30], and hybrid ARQ [43] introduce redundancy incrementally. However, fountain codes, rateless coding have been mostly used for bulk data transfer or non-real-time streaming. Hybrid ARQ has been mostly used in local wireless networks where the round-trip time is much smaller compared to real-time delay constraints. When the round-trip time is comparable to real-time delay constraints, incremental redundancy schemes degenerate to FEC. Many other sophisticated schemes such as multi-description coding [50] also use FEC to scale the video quality proportional to the bandwidth. While these schemes relax some of the above limitations of FEC, the fundamental limitation of small batch size is inherent to delay-sensitive applications.

3 Redundant Packet Transmission

We now describe the design of Redundant Packet Transmission (RPT), a simple variant of RT that sends fully redundant packets for delivering interactive video streams. We envision a scenario in which real-time video traffic and other traffic coexist, with no more than 50% of the traffic on a particular link being interactive, real-time traffic. We picked this scenario because it is representative

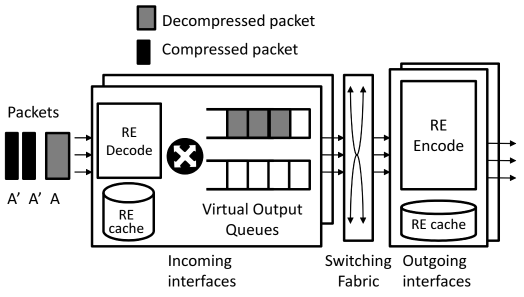


Figure 1: Redundant Packet Transmission in a redundancy elimination router.

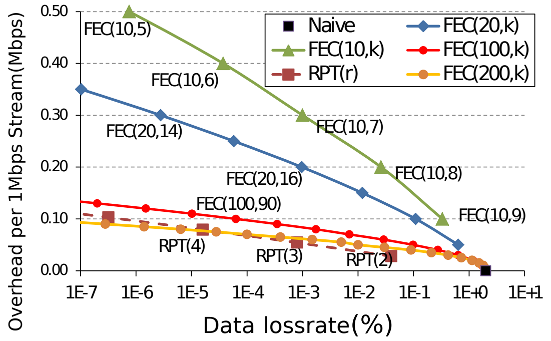


Figure 2: RPT and FEC under 2% random loss.

of forecasts of future network traffic patterns [3].

To simplify exposition, throughout this section, we assume that the RPT flows travel through a network with hop-by-hop Redundancy Elimination (RE) enabled [14]. Later, in §5, we explore RT in content-aware networks of various other forms. As stated earlier, we assume that packet losses happen only due to congestion.

We start by providing background on RE. We then describe our basic idea, followed by a description of key benefits and some comments on our approach.

3.1 Redundancy Elimination Background

In Anand et al's [14] design, RE is deployed across individual ISP links. An upstream router remembers packets sent over the link in a cache (each cache holds a few tens of seconds' worth of data) and compares new packets against cached packets. It encodes new packets on the fly by replacing redundant content (if found) with pointers to the cache. The immediate downstream router maintains an identical packet cache, and decodes the encoded packet. RE is applied in a hop-by-hop fashion.

RE encoding and decoding are deployed on the line cards of the routers as shown in Figure 1. Decoding happens on the input interface before the virtual output queue, and encoding happens on the output interface. The router's buffers (virtual output queues) contain fully decoded packets.

3.2 Basic Idea

As explained earlier, the basic idea of redundant packet transmission (RPT) is to send multiple copies of the same packet. If at least one copy of the packet avoids network loss, the data is received by the receiver. In current network designs, transmitting duplicate packets would incur large overhead. For example, if two duplicates of every original packet are sent, the overhead is 200% and a 1Mbps stream of data would only contain 0.33Mbps of original data. However, in networks with RE, duplicate copies of packets are encoded into small packets, and this overhead would be significantly reduced.

Figure 1 illustrates how RPT works with redundancy elimination. From the input link, three duplicate packets are received. The first packet is the original packet A, and the other two packets A', are encoded packets which have been "compressed" to small packets by the previous hop RE encoder. The compressed packet contains a reference (14 bytes in our implementation) used by the RE decoder of the next hop. At the incoming interface, the packets are fully decoded, generating 3 copies of packet A. They are then queued at the appropriate output queue. The figure illustrates a router that uses virtual output queuing. When congestion occurs, packets are dropped at the virtual output queue. Only packets that survive the loss will go through to the RE encoder on the output interface. When multiple packets survive the network loss, the first packet will be sent as decompressed, but the subsequent redundant packets will again be encoded to small packets by the RE encoder.

In this manner, multiple copies of packets provide robustness to loss and RE in the network reduces bandwidth overhead of additional copies.

3.3 Key Features

Next, we discuss three practically important properties of RPT: *high degree of robustness with low bandwidth overhead*, *ease of use and flexibility for application developers*, and *flow prioritization in the network*.

3.3.1 Low Overhead and High Robustness

As discussed in [14], the packet caches in the RE encoder and decoder are typically designed to hold all packets sent within the last tens of seconds. This is much longer than the timescale in which redundant packets are sent (∼60ms). Thus, all redundant packets sent by the application will be encoded with respect to the original packet. The extra bandwidth cost of each redundant packet is only the size of the encoded packet (43 bytes in our implementation[1].) The overhead of an extra redundant packet, therefore, is less than 3% for 1,500 byte packets, which is 7 to 17 times smaller than the typical FEC overhead for a Skype video call [20, 25].

To compare RPT with FEC, we model RPT and FEC under a 2% uniform random packet loss and analytically

[1] Our implementation does not encode IP and transport layer headers.

derive the data loss of a 1Mbps RPT and FEC streams in an RE network. Figure 2 shows the resulting overhead and data loss rate. All flows operate on a fixed budget but splits its bandwidth between original data and redundancy. The overhead (*y-axis*) is defined as the amount of redundancy in the stream, and the data loss (*x-axis*) as the percentage of data that cannot be recovered. RPT(r) denotes redundant streaming that sends *r* duplicate packets. FEC flows with various parameters are shown for comparison. FEC(n,k) denotes that *k* original packets are coded in to *n* packets. For FEC, we use a systematic coding approach (e.g. Reed-Solomon) that sends *k* original packets followed by $n-k$ redundant packets. While both schemes introduce redundancy, only the redundancy introduced by RPT gets minimized by the network unlike FEC which does not introduce redundancy in a way that the network understands; thus FEC over RE networks is identical in performance to FEC over traditional networks.

FEC schemes, especially with a small group size (*n*), incur large overheads, and are much less effective in loss recovery. For example, FEC(10,8), which adds 0.2 Mbps of redundancy, has similar data loss rates as RPT(2), which only adds 0.03Mbps of redundancy. FEC with group size (n=200) performs similar to RPT. However, it takes 2.4 seconds to transmit 200 1,500 byte packets at 1Mbps. This violates timing constraints of real-time communications because a packet loss may only be recovered 2.4 seconds later in the worst case. Thus, in practice, RPT provides high robustness against packet loss at low overhead.

3.3.2 Ease of Use and Control

Application developers can easily tailor RPT to fit their needs. Three unique aspects of RPT help achieve this property:

1) Detailed parameter tuning is not necessary.
2) RPT allows per-packet redundancy control.
3) Delay and redundancy are decoupled.

Ease of parameter selection: With FEC, the sender has to carefully split its bandwidth between original and redundant data in order to maximize the video quality. If the amount of redundancy is larger than the amount of network loss, the stream tolerates loss. However, this comes at the cost of quality because less bandwidth is used for real content. If the amount of redundancy is too low, the effect of loss shows up in the stream and the quality degrades. This trade-off is clear in FEC(10,k)'s performance in Figure 2. Determining the optimal parameters for FEC is difficult and adapting it to changing network conditions is even more so [29].

A unique aspect of RPT is that even though the actual redundancy at the sender is high, the network effectively reduces its cost. Therefore, the sender primarily has to ensure that the amount of redundancy (*r*) is high enough to tolerate the loss and worry much less about its cost, which makes RPT simple and easy to use. We show in §6.3 that only small amount of redundancy ($r = 3$) is good enough for a wide range of loss rates (1% to 8%), and a sub-optimal overshoot (i.e. unnecessary, extra redundancy) has very little impact on actual video quality.

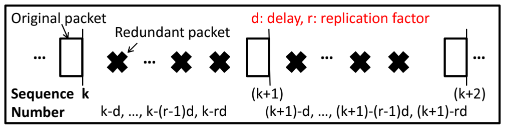
Packet-by-packet redundancy control: RPT introduces redundancy for each packet as opposed to groups of packets, enabling packet-by-packet control of the extent of redundancy. More important packets, e.g., those corresponding to I-frames, could simply be sent more repeatedly than others to increase robustness. In essence, RPT enables fine-grained unequal error protection (UEP) [33]. Thus, RPT is simple to adapt to application-specific needs and data priorities.

Each encoded packet can be viewed as an implicit signal to the network. Importance of the data is encoded in the number of encoded packets, $r-1$. When an original packet gets lost, routers try to resend the original packet when the signal arrives. As such the network tries up to *r* times until one original copy of the packet goes through.

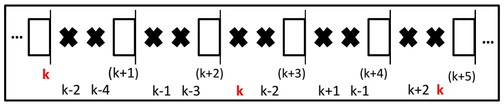
Decoupling of delay and redundancy: Unlike FEC, RPT separates the redundancy decision from delay. FEC schemes closely tie timing with the encoding since the receiver cannot reconstruct lost packets until the batch is complete. In contrast, RPT accommodates timing constraints more easily. For example, sending 3 redundant packets spaced apart by 5 ms is essentially asking every router to retry up to 3 times every 5 ms to deliver the original packet. This mechanism lends itself to application specific control to meet timing constraints. We further discuss the issues in controlling delay in §3.4.

3.3.3 Flow Prioritization

A unique property of RPT-enabled traffic is that it gets preferential treatment over other traffic under lossy conditions. RPT flows do not readily give up bandwidth as quickly as non-RPT flows. This is because for RPT flows packet losses do not directly translate into less bandwidth use due to "deflation" of redundant packets; subsequent redundant packets cause retransmission of the original packet when the original packet is lost. Therefore, *RPT flows are effectively prioritized in congested environments.* As a result, RPT could get more share at the bottleneck link. We believe that this is a desirable property for providing stronger guarantees about the delivery rate of data, and analyze this effect in §6. However, this preferential treatment may not be always desirable. In case where fair bandwidth-sharing is desired, RPT is flexible enough to be used with existing congestion control mechanisms while retraining its core benefits. In §4, we provide an alternative solution that retains other two benefits of RPT except flow prioritization.



(a) Sequence of packets sent by RPT(r)



(b) Sequence of packets sent by RPT(3) with d=2

Figure 3: Sequence of packets sent by RPT

3.4 Scheduling Redundant Packets

We now discuss detailed packet sequencing, i.e. how RPT interleaves redundant packets with original packets. Each original packet is transmitted without any delay, but we use two parameters to control the transmission of redundant packets: redundancy (r) and delay (d).

Figure 3(a) shows the packet sequence of an RPT(r) flow. Original packets are sent without any delay, and $r-1$ redundant packets are sent compressed in between two adjacent original packets. Thus, compared to a non-RPT flow of the same bitrate, r times as many packets are sent by a RPT(r) flow.

The delay parameter (d) specifies the number of original packets between two redundant packets that encode the same data. The first redundant packet of sequence number n is sent after the original packet of sequence number $(n+d)$. If the loss is temporally bursty, having a large interval between two redundant packets will help. However, extra delay incurs extra latency in recovering from a loss. So, delay (d) can be adjusted to meet the timing requirements of applications.

Figure 3(b) shows an example with $r = 3$ and $d = 2$. Three copies of packet k is sent, each spaced apart by two original packet transmissions. In §6.3, we evaluate RPT's sensitivity to parameter selection.

3.5 Comments on RT/RPT

Is this link-layer retransmission? Conceptually, RT is similar to hop-by-hop reliability or link-layer retransmission. However, RT fits better with the end-to-end argument-based design of the Internet by giving endpoints an elegant way to control the retransmission behavior inside the network. In contrast, hop-by-hop reliability schemes make it hard for applications to control the delay or to signify the relative importance of data. Similarly, in a naive hop-by-hop retransmission scheme, packets are treated equally and can be delayed longer than the application-specific limit. RT exploits network's content-awareness and provides a signaling mechanism on top of such networks to achieve robustness against packet loss.

Why not make video codecs resilient? In the specific context of video, prior works have proposed making video codecs more resilient to packet loss. Examples include layered video coding [46], H.264 SVC, various loss concealment techniques [55] and codecs such as ChitChat [56]. However, greater loss resilience does not come for free in these designs; these designs typically have lower compression rate than existing schemes or incorporate redundancy (FEC) in order to reconstruct the video with arbitrary loss patterns. Also they are often more computationally complex than existing approaches, which makes them difficult to support on all devices [55].

Our scheme is agnostic to the choice of video codec and the loss concealment schemes used. Of course, the exact video quality gains may differ based on the loss rates, loss patterns and codec used.

How does RT compare to more sophisticated coding? Many sophisticated video coding schemes, such as UEP [33], priority encoding transmission [12], and multiple description coding [23, 50], typically use FEC (or Reed-Solomon codes) as a building block to achieve graceful degradation of video quality. Similarly, we believe that RT can be used as a building block to enable more sophisticated schemes. For example, one can send more important blocks of bits within a stream multiple times. Furthermore, since RE networks also eliminate sub-packet level redundancy, a partially redundant packet may also be used. We leave details of such techniques as future work. In this work instead, we focus on understanding the core properties of RT by comparing a basic form of RT with the most basic use of FEC.

What about wireless errors? We do not yet know if RT/RPT can be extended to protect against categories of losses other than those due to congestion, e.g., partial packet errors due to interference and fading. We do note that there a variety of schemes that aim to provide robust performance in such situations, some with a focus on video (e.g., the schemes in [38, 39, 53] for wireless links). However, RT/RPT's explicit focus on congestion losses means that our approach is complementary to such schemes. In our technical report [31], we discuss how RT can happily coexist with such schemes.

4 RPT with Congestion Control

As explained earlier, RPT flows are effectively prioritized in congested environments[2]. However, in some environments, fair bandwidth sharing may be more desirable. In such cases, the sending rate should adapt to the network conditions to achieve a "fair-share". To meet this goal, we apply TCP friendly rate control [28] (TFRC) to RPT flows. However, this raises surprisingly subtle problems regarding the transmission rate and loss event rate estimation that are germane to TFRC. We describe these challenges and our modifications to TFRC below.

[2]We further verify this later in §6.4.

Packet transmission: In TFRC for RPT, we calculate the byte transmission rate from the equation just as the original TFRC. Note that RPT(r) must send an *original* packet and $r-1$ duplicates. To match the byte sending rate, we adjust the length of the packet so that equal number of bytes are sent by TFRC RPT as the original TFRC in calculating the throughput. Thus, given a computed send rate, TFRC RPT(r) sends r times as many packets. Note that each packet, original or duplicate, carries an individual sequence number and a timestamp for TFRC's rate calculation purposes.

Loss event rate estimation: In the original TFRC, the sending rate is calculated given the loss event rate p, where loss event rate is defined as the inverse of the average number of packets sent between two loss events. A loss event is a collection of packet drops (or congestion signals) within a single RTT-length period.

Ideally, we would want TFRC RPT(r) to have the same loss event rate as the original TFRC, as that would also make TFRC RPT obtain a TCP friendly fair share of bandwidth. However, the observed loss event rate for RPT depends on the underlying packet loss pattern. For ease of exposition, we look at the two extremes of loss patterns: one that is purely random and the other that is strictly temporally correlated.

Purely random packet drops may occur in a link of a very high degree of multiplexing. On the other hand, in a strictly temporal loss pattern, losses occur during specific intervals. One might see such a loss pattern when cross traffic fills up a router queue at certain intervals. In reality, the two patterns appear inter-mixed depending on source traffic sending patterns and the degree of multiplexing.

Next, we discuss how the two loss patterns impact loss event rate estimation and the transmission rate:

- *Uniform random packet loss:* In this setting, TFRC RPT(r) behaves in a TCP friendly manner without any adjustment to loss estimation. This is because the number of packets sent between two loss events does not change even though the packet sending rates change.
- *Temporal packet loss:* In this setting, packets are lost at specific times. During the time between loss, TFRC RPT(r) sends r times as many packets. Thus, the observed loss event rate for TFRC RPT(r) is only $\frac{1}{r}$ of that of the original TFRC. Therefore, TFRC RPT would send more traffic.

Adjusting the loss event rate: As stated earlier, in practice, the two extreme patterns appear inter-mixed. We therefore want to choose an adjustment factor α so that when the loss event rate is adjusted to α times the measured loss event rate p, TFRC RPT(r) is TCP friendly. As seen in the two extreme cases, α has values 1 for uniform random losses and r for temporal losses, respectively. So, in practice, α should be between 1 and r to achieve exact TCP-friendliness. A larger value of α makes the TFRC-RPT react more aggressively to congestion events, and smaller value less aggressive than a TCP flow. This means, even in the worst case, α can be r times off from the value which achieves exact TCP-friendliness. In this case, a TFRC-RPT flow would have performance similar to $\sqrt{r}$ many TFRC flows because the TCP-friendly rate is inversely proportional to $\sqrt{p}$. Therefore, even an incorrect value of α would still make TFRC RPT friendly to a group of TCP connections and still react to congestion events. In practice, we find in §6 that with TFRC-RPT(3), $\alpha = 1.5$ closely approximates the bandwidth share of a single TCP flow under wide range of loss rates and realistic loss patterns.

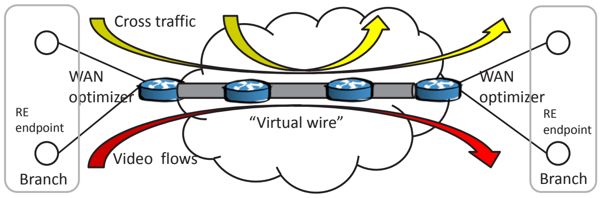


Figure 4: Typical Deployment of WAN optimizers

As such, RT is flexible enough to allow users to adjust the degree of reactivity to congestion events while being highly robust. Regular RPT does not react to congestion events, and can be used to prioritize important flows. TFRC RPT reacts to congestion events and the reaction degree can be controlled by the parameter α.

5 RPT in Various Networks

So far, we have explored RPT on hop-by-hop RE networks as a special case of redundant packet transmission. Here, we look at other deployment scenarios for content-aware devices as well as other content-aware designs.

Corporate networks: WAN optimization is the most popular form of RE deployment in the real world. In a typical deployment, WAN optimizers are placed at branch and main offices or between data-centers to reduce the traffic between them. Example deployments include 58+ customers of Riverbed [10] and Cisco's worldwide deployment to its 200+ offices [8]. While we envision RPT being used in future networks where content-aware devices are widely deployed, RPT can be deployed immediately in such settings.

As shown in Figure 4, these sites have low bandwidth connections using leased line or VPN-enabled "virtual" wires. ISPs offering VPN services typically provide bandwidth and data delivery rate (or packet loss) guarantees as part of their SLA [1, 6]. In practice, their loss rate is often negligible because ISPs provision for bandwidth [24]. [3] Thus, the use of VPN and WAN optimizers effectively creates reliable RE "tunnels" on which RPT can operate. Important, real-time data can be sent with redundancy,

[3]Sprint's MPLS VPN [6] had a packet loss rate of 0.00% within the continental US from Mar 2011 to Feb 2012.

	RPT(3)	FEC(6,5)
Overhead	9%	22%
Data loss rate	8.0 x 10^{-6}	1.9 x 10^{-3}

Table 1: Comparison of FEC and RPT over CCN

and compete with other traffic when entering this tunnel. Packets will be lost when the total demand exceeds the capacity of the tunnel, but RPT flows will have protection against such loss. We evaluate this scenario in §6.2.

An alternative is to use traditional QoS schemes such as priority queuing. However, this typically involves deploying extra functionalities including dynamic resource allocation and admission control. For businesses not willing to maintain such an infrastructure, using RPT on and existing RE-enabled VPN would be an excellent option for delivering important, time-sensitive data.

Partial deployment: Not all routers in a network have to be content-aware to use RPT. The requirement for "RPT-safety" is that RE is deployed across bandwidth-constrained links [4], and non-RE links are well provisioned. This is because non-RE links end up carrying several duplicate packets. When such links are of much higher capacity, RPT causes no harm. Otherwise, it impacts network utilization and harms other traffic. In §6.2, we explore both cases through examples, and show how the network utilization and the other traffic on the network are impacted when RPT is used in an "unsafe" environment.

To ensure safe operation of RPT, one can detect the presence of RE on bandwidth-constrained links, and use RPT only when it would not harm other traffic. In this section, we outline two possible approaches for this, but leave details as a future work. One approach is to use end-point based measurement: for example, Pathneck [34] allows detection of bottlenecks based on available bandwidth. It sends traceroute packets in between load packets and infers (multiple) bottleneck location(s) from the time gap between returned ICMP packets. Similar to this, we can send two separate packet trains: one with no redundancy and the other with redundancy r but with the same bitrate. If all bandwidth constrained links are RE-enabled and RPT is safe to use on other links, the packet gap would not inflate on previously detected bottlenecks and the redundant packet trains would not report different bottleneck links. Another way is to use systems, such as iPlane [45] and I-path [47], which expose path attributes (e.g. available bandwidth) to end-hosts. These systems can easily provide additional information such as RE-functionality for end-hosts to check for RPT safety.

RPT over CCN: RPT also can be integrated with a broad class of content-aware networks, including CCN [37] and SmartRE [15]. In our technical report [31], we explore discuss how RPT can work atop SmartRE and wireless networks. Here, we focus on applying RPT to CCN.

[4]This matches the common deployment scenario for RE [14, 54].

Quality	Excellent	Good	Fair	Poor	Bad
PSNR (dB)	> 37	$31 \sim 37$	$25 \sim 31$	$20 \sim 25$	< 20

Table 2: User perception versus PSNR

In CCN, data consumers send "Interest" packets, and the network responds with at most one "Data" packet for each Interest. Inside the network, each router caches content and eliminates duplicate transfers of the same content over any link. CCN is designed to operate on top of unreliable packet delivery service, and thus Interest and Data packets may be lost [37].

We now compare RPT and FEC in CCN. Suppose real-time data is generated continuously, say k packets every 100 ms, and RTT is large. In an FEC-equivalent scheme for CCN, the content publisher would encode k data packets and add $(n-k)$ coded data packets, where $n > k$. The data consumer would then generate n Interest packets for loss protection. The receiver will be able to fully decode the data when more than k Interest and Data packets go through. However, up to n Interest/Data pairs will go through the network when there is no loss. In contrast, RPT does not code data packets, but generates redundant Interest packets. This obviously provides robustness against Interest packet loss. Moreover, when a Data packet is lost, subsequent redundant Interest packet will re-initiate the Data transfer. Since Interest packets are small compared to Data and duplicate Interests do not result in duplicate transfers of the Data, the bandwidth cost of redundancy is minimal. In RPT, at most k Data packets will be transferred instead of n in the FEC scheme.

To demonstrate the benefit more concretely, we take the Web page example from CCN and compare RPT and FEC over CCN. In the CCN-over-jumbo-UDP protocol case [37], a client generates three Interest packets (325 bytes) and receives five 1500-byte packets (6873 bytes) to fetch a Web page [37]. To compare RPT and FEC, we assume in RPT a redundancy parameter of 3 is used and in FEC the server adds one packet to the original data. Table 1 shows the overhead and data loss rate of each scheme at the underlying loss rate 2%. The data loss rates is the amount of data that could not be recovered. Even though the overhead of RPT is only 41% of that of FEC, it's data loss rate is 240 times better. To achieve equal or greater level of robustness than RPT($r = 3$), FEC has to introduce 11 times the overhead of RPT($r = 3$).

6 Evaluation

In this section, we answer four specific questions through extensive evaluation:

(*i*) **Does RPT deliver better video quality?** How well does it work in practice?

In §6.2, we show that RPT provides high robustness and low bandwidth overhead, which translate to higher quality for video applications.

(*ii*) **Is RPT sensitive to its parameter setting, or does it require fine tuning of parameters**?

In §6.3, we show that, unlike FEC, RPT is easy to use since careful parameter tuning is not necessary, and delay can be independently controlled with the delay parameter.

(*iii*) **How do RT flows affect other flows and the overall network behavior**?

In §6.4, we demonstrate that RT flows are effectively prioritized over non-RT flows on congested links and may occupy more bandwidth than their fair-share.

(*iv*) **Can we make RT flows adapt to network conditions and be TCP-friendly**?

We show in §6.5 that RT can also be made TCP-friendly, while retaining the key benefits.

6.1 Evaluation Framework

We use a combination of real-world experiments, network measurements and simulations. We implemented an RE encoder and decoder using Click [41], and created a router similar to that of Figure 1. Using this implementation, we create a hop-by-hop RE network in our lab as well as in Emulab. These serve as our evaluation framework.

We use implementation-based evaluation to show the overall end-to-end performance of RPT, and simulations to unravel the details and observe how it interacts with other cross traffic. To obtain realistic packet traces and loss patterns from highly multiplexed networks, we performed active measurements to collect real-world Internet packet traces. We also created background traffic and simulated RPT and FEC flows in a hop-by-hop RE network using the ns-2 simulator. These video flow packet traces are then fed into *evalid* video performance evaluation tool [40] to obtain the received video sequence with loss. For video, we used the *football* video sequence in CIF format, taken from a well-known library [7]. We used H.264 encoding with 30 frames per second. I-frames were inserted every second and only I- and P-frames were used to model live streams.

RE implementation: We implemented the Max-Match algorithm described in [14]. We further modified it to only store non-redundant packets in the packet cache. Therefore, sending redundant packets does not interfere with other cached content. We use a small cache of 4MB. The implementation of the encoder encodes a 1500 byte fully redundant packet to a 43 byte packet[5]. We also implemented RE in ns-2.

Evaluation metric: We use the standard Peak-to-Signal-to-Noise Ratio (PSNR) [52] as the metric for the video quality. PSNR is defined using a logarithmic unit of dB, and therefore a small difference in PSNR results in visually noticeable difference in the video. The MPEG committee reportedly uses a threshold of PSNR = 0.5dB to test the significance of the quality improvement [52]. Typical values for PSNR for encoded video are between 30 and 50 dB. Table 2 maps the PSNR value to a user perceived video quality [40].

[5]We do not compress network and transport layer headers. Thus, the packet may be compressed even further in practice.

6.2 End-to-end Video Performance

In this section, we evaluate the end-to-end performance of RPT and examine key characteristics.

Experimental setting: First, we use our testbed based on our hop-by-hop RE implementation, and create a streaming application that uses redundant packet transmission. We create a topology where an RE router in the middle connects two networks, one at 100Mbps and the other at 10Mbps. To create loss, we generate traffic from the well-connected network to the 10Mbps bottleneck link.

We generate a 1Mbps UDP video stream and long-running TCP flows as background traffic. We adjust the background traffic load to create a 2% loss on the video flow. We then compare the video quality achieved by RPT, Naive, and FEC that use the same amount of bandwidth. We use RPT that has 6% overhead ($r = 3, d = 2$), and FEC(10,9) with 10% overhead, which closely match in latency constraints with comparable overhead. Naive uses UDP without any protection.

Figure 5 shows the sending rate and the received data rate after the loss. The RPT and FEC senders respectively use about 6% and 10% of their bandwidth towards redundancy, while the Naive sender fully uses 1Mbps to send original data. The sending rates of the three senders are the same, within a small margin of error (1%). The Naive receiver loses 2% of the data and receives 0.98Mbps because of the loss. The FEC receiver only recovers about 66% of the lost data due to the bursty loss pattern. On the other hand, the RPT receiver receives virtually all original data sent. Note that only the amount of redundancy has slightly decreased. This is because when an original packet is lost, a subsequent redundant packet is naturally expanded inside the network.

As a result, the RPT flow gives much higher video quality. Figure 6 shows a snapshot of the video for RPT and Naive flows. Table 3 shows the video quality of an encoded video and the received video. The encoded video column shows the quality of video generated at the sender before packet loss. When RPT and FEC are used, the encoded video quality is slightly lower because of the bandwidth used towards redundancy. However, **because the RPT flow is highly robust against loss, it provides the best video streaming quality** (1.8 dB better than FEC and almost 6dB better than Naive).

RE-enabled Corporate VPN of §5 is the most common deployment scenario of RT in today's networks. To demonstrate the feasibility of this scenario, we set up a network of four routers in Emulab [26] and created an RE-enabled VPN tunnel that isolates the traffic be-

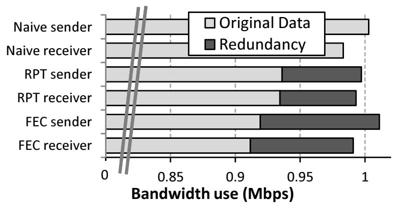
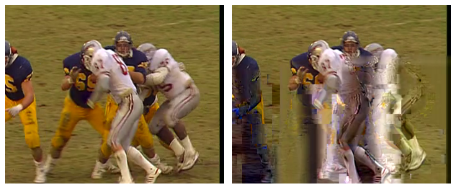


Figure 5: Bandwidth use



(a) RPT flow (b) Naive flow

Figure 6: Snapshot of the video

	Encoded	Received
RPT	37.3 dB	**37.1 dB**
FEC	36.9 dB	**35.3 dB**
Naive	37.5 dB	**31.4 dB**

Table 3: Average video quality (PSNR)

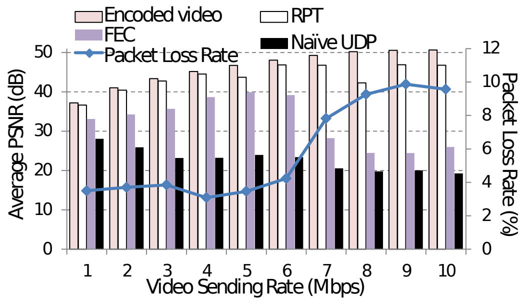


Figure 7: Video quality and loss rate for real traces

tween two remote offices similar to that of Figure 4. The physical links between two remote offices have 100Mbps capacity, and carries traffic from other customers. We generate cross traffic over the physical links that carries the VPN traffic so that the physical links experience congestive loss. We allocate 5Mbps of bandwidth to the VPN-enabled “virtual” wire, which is emulated using the priority queuing discipline from the Linux kernel’s traffic control module. We introduced a 1Mbps video traffic and 5 TCP connections between the two remote offices, and compare RPT(3) and FEC(10,9) whose bandwidth overhead best matches to that of RPT(3), while adhering to the latency constraint. The video stream experiences a loss rate of around 2.7% and the tunnel’s link utilization was nearly 100% in both cases. The resulting PSNR of the RT flow and FEC were 37.1dB and 34.1dB respectively. This result shows that RT also works well on the most common form of today’s content-aware networks.

Real traces: To study the performance of RPT in a realistic wide-area setting, we collected a real-world packet trace. We generated UDP packets from a university in Korea to a wired host connected to a residential ISP in Pittsburgh, PA. The sending rate was varied from 1Mbps to 10Mbps, each run lasting at least 30 seconds. The round-trip-time was around 250ms, which indicates that retransmissions would violate the timing constraint of an interactive stream.

Assuming that the packet loss rates do not change significantly with RPT[6], we apply the loss pattern obtained from the measurement to an RPT flow, an FEC flow and a Naive UDP flow. For RPT, we use a redundancy parameter of $r = 3$ and a delay parameter $d = 2$. For FEC, we choose the parameters so that the overhead matches closest to that of RPT, while the additional latency incurred by FEC at the receiver does not exceed 150ms, which results in different parameters for different sending rates. Figure 7 shows the video quality for each scheme. The solid line shows the packet loss rate. The `Encoded video` bar shows the ideal PSNR without any loss when all the bandwidth is used towards sending original data, presented as a reference. As the sending rate increases, the quality of the encoded video increases. However, the loss rate from Korea to U.S. was 3.5% at 1Mbps but increased to 9.8% at 10Mbps as the sending rate increases. Because of the high loss rate, the naive UDP sender performs poorly (PSNR well under 30dB). FEC achieves better performance than naive, but much worse than RPT especially under high loss rates. In contrast, **RPT gives the best performance, closely following the quality of the encoded video until the loss rate is about 8%**. Even at the higher loss rates, the impact on quality is much less than the FEC scheme. This is because RPT gives much better protection against loss than FEC at similar overhead. [7]

6.3 Parameter Selection and Sensitivity

In this section, we provide an in-depth performance evaluation of RPT. In particular, we compare RPT and FEC’s parameter sensitivity using simulations that produce packet loss patterns of highly multiplexed networks with realistic cross traffic.

Simulated RE Network: We use the ns-2 simulator to create a realistic loss pattern by generating a mix of HTTP and long-running TCP cross traffic. We use a dumbbell topology with the RE-enabled bottleneck link capacity set to 100Mbps, and simulate a hop-by-hop RE network and RPT flows. We generate 100 long-running TCP flows and 100 HTTP requests per second. We used the packmime [22] module to generate representative HTTP traffic patterns. We also generate ten video flows each having

[6]We later verify this and see how RPT affects the loss rate in §6.4.

[7]Large drop in PSNR at 8 Mbps is an artifact of the video’s resolution being too small compared to its encoding rate and a particular pattern of bursty data loss. When the video compression gets nearly lossless, even a small data loss causes PSNR to drop sharply. In addition, two original packets that are close together in sequence were lost by coincidence in 8Mbps RPT. This had a more detrimental effect on the PSNR value because the lost data belonged to the same video frame. The actual data loss rate of the 8Mbps RPT flow was 0.165%, which is less than 0.174% of the 9Mbps RPT flow.

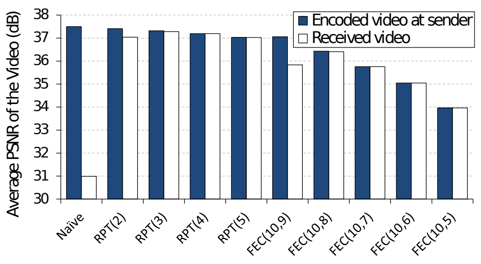


Figure 8: RPT's performance is much less sensitive to its parameter setting.

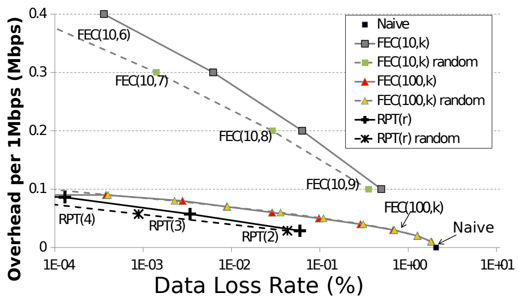


Figure 9: Percent data loss rate and overhead: RPT greatly outperforms FEC with small group size.

1Mbps budget regardless of the loss protection scheme it uses. The results presented are averages of ten runs with each simulating five minutes of traffic. We first look at the final video quality seen by the end receiver under different parameter settings, and then analyze the underlying loss rate and overhead.

How do RPT flows and FEC flows perform with different redundancy parameters? For RPT, we vary the redundancy parameter r from 2 to 5, while fixing the delay parameter d to 2. For FEC, we use a group size $n = 10$ to meet the latency constraints and vary the number of original data packets k from 5 to 9.

Figure 8 shows the quality of the video seen by the receiver compared to the encoded quality at the sender. The result shows that **RPT performs better than FEC's best, and its performance is stable across different parameter settings.** In contrast, FEC's performance is highly sensitive to the parameter selection. Therefore, with FEC, the sender has to carefully tune the parameter to balance the amount of redundancy and encoding rate.

Figure 9 shows the underlying data loss rate and overhead of the video flows. The x-axis shows the data loss rate in log-scale, and the y-axis shows the amount of overhead. All video flows experience $\sim$2% packet loss. For comparison, the performance of RPT and two FEC families (n=10, 100) under uniform random loss (dotted lines) are also shown. RPT(4)'s data loss rate is several orders of magnitude lower than the loss rate of FEC(10,9) whose overhead is similar. RPT(4) even performs better than FECs with large group size, such as FEC(100,91), whose latency exceeds the real-time constraint. FEC(10,7) achieve similar data loss rate to RPT(3) but has 6 times the overhead, which translated to 2dB difference in PSNR.

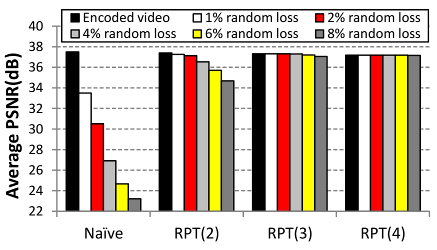


Figure 10: Video quality under 1 to 8% loss. RPT(3) steadily delivers high quality even under high loss.

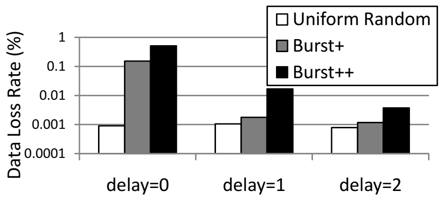


Figure 11: Bursty loss increases the data loss especially when the delay parameter is small.

The gap between the uniform loss and actual loss lines in Figure 9 represents the effect of bursty loss performance. FEC(10,k) and RPT show a relatively large gap between the two lines. Analyzing the underlying loss pattern, we observe that within a group of 10 packets, losses of 2 to 4 packets appear more frequently in the actual loss pattern. This shows that the underlying traffic is bursty. On the other hand, loss bursts of more than 5 occur less frequently in the actual pattern because TCP congestion control eventually kicks in.

We now show how the parameter should be set in RPT.

How should we choose parameters in RPT? To answer this question, we study how loss rate and burstiness of loss affect the performance of RPT. First, we use the random loss pattern and vary the packet loss rate from 1% to 8%. For RPT, we vary the redundancy parameter from 2 to 4, but fix the delay parameter at 2. For each loss rate, the average PSNR of a naive sender and an RPT sender is shown in Figure 10. It shows that video quality of RPT(3) is virtually immune to a wide range of packet losses; the average PSNR for RPT(3) under 8% loss only decreased by 0.25dB compared to the zero-loss case. We, therefore, use $r = 3$ in the rest of our evaluation.

Second, we look at the role of the delay parameter under bursty loss. For reference, we generate a 2% random loss, which on average has 1 lost packet every 50 packets. We then create bursty loss patterns by reducing the number of packets between losses by up to 15 and 35, while keeping the average loss rate the same. The three cases are named as `Uniform random`, `Burst+`, and `Burst++` respectively. Figure 11 shows the data loss rate of RPT with different delay parameters under the three loss conditions. An increase in burstiness negatively impacts the data loss rate, but as delay is increased

	FEC(10,6)	FEC(100,92)	RPT(3)
No sender buffering	240ms	2400ms	60ms
Sender buffering	180ms	1300ms	-

Table 4: Maximum one-way delay of RT and FEC

the negative impact is decreased. In general, a large delay parameter gives more protection against bursty loss, but incurs additional latency. **We use $d = 2$ and $r = 3$ for RPT because of its superior performance in wide range of loss conditions**.

How much latency is caused by RPT and FEC flows? A loss might be recovered by subsequent redundant packets; here we quantify the delay in this. In RPT(r) with delay d, the receiver buffer must hold $d \cdot r$ packets. So the delay in RPT is $d \cdot r \cdot intv$, where $intv$ is the interval between packets. The RPT sender needs no additional buffering, as it transmits the packet as soon as a packet is generated from the encoder. In FEC, two alternatives exist where one does sender buffering to pace packets evenly and the other doesn't but further delays the transmission of redundant coded packets [31]. Table 4 shows the delay caused by RPT and FEC for 1Mbps RPT(3) with $d = 2$ and FEC streams that exhibit similar data loss rate from Figure 9. We see that **RPT gives a significantly lower delay than FEC schemes of similar strength in loss protection**, and FEC(100,k) is not suitable for delay-sensitive communication.

How do RPT and FEC flows perform under extreme load? One might think that in a highly congested link with a high fraction of RPT traffic, RPT flows would constantly overflow the buffer and the performance would drop. To create such a scenario with increased traffic load, we vary the fraction of video traffic in the link from 10% to 90%, while keeping the number of background TCP connections and bottleneck bandwidth the same. Detailed evaluation is provided in [31]. In summary, we find that **RPT flows achieve close-to-ideal video quality, and better quality compared to the best FEC scheme** in all cases (10% to 80%) except for one very extreme case (90%) with very heavy cross traffic creating loss rate $> 10\%$. The extreme case we portray in our experiment is unlikely to occur in practice for two reasons: 1) The loss rates in practice are likely to be much lower. 2) Even the aggressive estimate suggests that no more than 15% of traffic in future will be real-time in nature [3].

6.4 Impact of RPT on the Network

We now examine the effect of RPT flows on other cross traffic and the network. For this evaluation, we use the same simulation setup and topology described in §6.3.

How do other TCP flows perform? We look at the impact on two different types of TCP flows: long-running TCP flows and HTTP-like short TCP flows.

To evaluate the *impact on long-running TCP*, we send

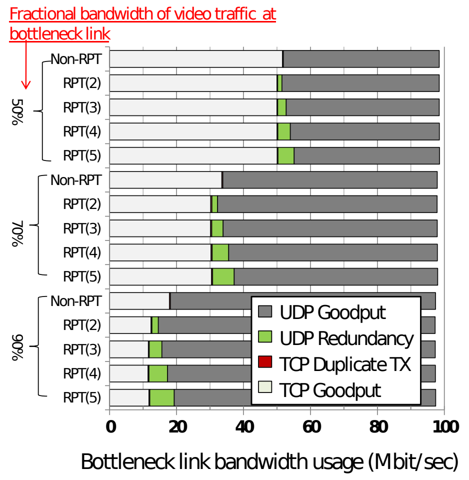


Figure 12: Breakdown of bottleneck link utilization: RPT flows do not impact the link utilization. RPT flows are prioritized over competing TCP flows.

100 long-running TCP flows and a varying number of video flows to vary the fraction of video traffic on the bottleneck link (from 10 to 90%). We also vary the redundancy parameter from 0 (Non-RPT) to 5. We use a small router buffer size of $\frac{2 \cdot B \cdot RTT}{\sqrt{100}}$ [16] to maximize the negative impact.

Figure 12 shows the bottleneck link traffic decomposition in four categories[8]: UDP goodput, UDP redundancy, TCP duplicate, TCP goodput. UDP goodput is the bandwidth occupied by the original packet and the UDP redundancy represents the bandwidth occupied by the compressed packets. TCP goodput represents packets contributing to application throughput, and TCP duplicate Tx shows the amount of duplicate TCP packets received.

We observe that **1) the bandwidth utilization is not affected by RPT, and 2) RPT flows are effectively prioritized over non-RPT TCP flows.** In all cases the bottleneck bandwidth utilization was over 97.5%; TCP fills up the bottleneck even if the router queue is occupied by many decompressed redundant packets. TCP throughput, on the other hand, is impacted by the RPT flow especially when the network is highly congested; e.g. in the 90% video traffic case (bottom-most bars), TCP goodput (white region) decreases when RPT is used. This is because when RPT and non-RPT cross traffic competes, even though they experience the same underlying packet loss rates, for RPT flows packet loss do not directly translate into throughput loss. With redundant transmissions, the network recovers the loss through subsequent uncompressed redundant packets, effectively prioritizing the RPT flows.

To see the *impact on HTTP-type short flows*, we look at how RT changes the response time of short flows under

[8]Only a subset of results (video traffic occupying 50% to 90% of bottleneck) shown for brevity, but all cases confirm the same results.

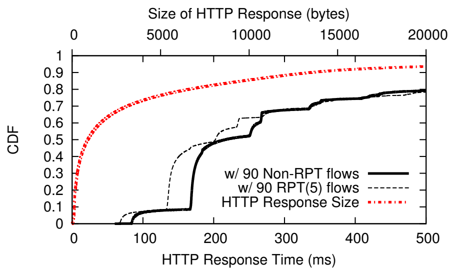


Figure 13: Response time and size for short HTTP flows (long flows omitted for clarity).

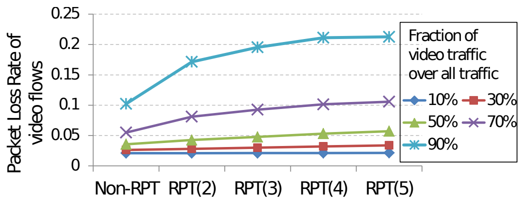


Figure 14: Impact on loss rate due to RPT flows.

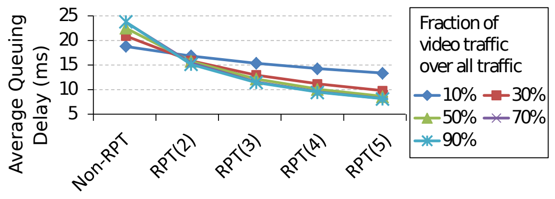


Figure 15: Queuing delay is reduced with RPT flows.

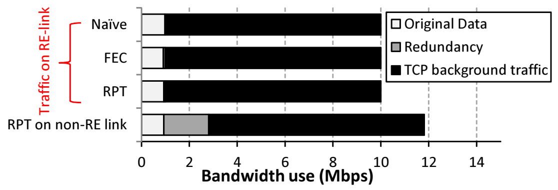


Figure 16: *No Harm:* Bandwidth use on a non-RE link in the no harm case.

the setup described in §6.3. Figure 13 shows the CDFs of the size of the response messages, and the response times. To highlight the difference, we only show response times when 90% of the traffic is video and RPT(5) is used, but the trend is visible across all cases.

The response time for short flows decreases in the presence of RPT flows. Since redundant packets in the queue are compressed when they are sent out, the service rate of the queue increases with RT. Therefore the queuing delay is reduced, which results in a decrease in the response time. However, for larger flows (tail end of the figure) the response time actually increases, as they behave more like long-running TCP flows, which obtain less throughput under congestion (Figure 12).

How does network behavior change with RPT flows? There are subtle changes in loss rate and queuing delay.

Loss rate: Figure 14 shows the packet loss rate of UDP video flows at the bottleneck router with varying amount of RPT traffic. **When the fraction of video traffic is moderate, adding more redundancy has little impact on the underlying packet loss of the video flow.** This is because while redundant packets increase the load, they also increase the service rate of the link. However, we observe that when RPT flows dominate the bottleneck link, the underlying loss rate for video flows goes up as redundancy increases. The underlying reason for increased loss is that under congestion RPT flows compete with each other for bandwidth when most of the traffic is from RPT flows. However, in §6.3, we saw that even under such extreme conditions RPT performs better than FEC.

Queuing Delay: Figure 15 shows the average queuing delay with varying redundancy parameters and varying number of RPT flows. **Redundant packets decrease queuing delay.** This is because redundant packets appear as decompressed at the router queue, but are sent out compressed at the bottleneck link. Therefore, as the number of redundant packets increase, service rate becomes faster.

What's the impact of partial content-awareness? In §5, we noted that RT may cause harm in partially content-aware networks and should be used only after detecting RT-safety. Here, demonstrate both the *no-harm* and the *harm* case, and quantify the impact using our experimental testbed, which has a 10Mbps and a 100Mbps RE link.

To demonstrate *no-harm*, we disabled the RE encoder on the non-bottleneck 100Mbps links of our testbed. We then generated an RPT flow and TCP background flows through this 100Mbps link and the 10Mbps RE-enabled link. Figure 16 shows the traffic on both links. The RPT flow occupying 1Mbps on an RE-enabled bottleneck link introduces almost 2Mbps of overhead (`redundancy`) on the non-RE 100Mbps link. However, this causes no harm since the 100Mbps link is not bandwidth constrained.

To demonstrate *harm*, we reduced the capacity of the non-RE link to 15Mbps, and introduced four 1Mbps video flows and a TCP flow. Table 5 compares bandwidth use on the 10Mbps RE link when the video flows are sent with and without redundancy. When there's no redundancy (`Non-RPT`), the link utilization of the 10Mbps link is 97%. When RPT(3) is used, the 4Mbps video flows occupy 11.2Mbps on the non-RE link. This shifts the bottleneck to be the 15Mbps non-RE link, which forces the 10Mbps RE-link and the network to be under-utilized at 76%. This verifies that in a partial deployment setting, detecting RT-safety is important as discussed in §5.

6.5 TCP-Friendly RPT

RPT flows do not give up bandwidth as easily under congestion. In §4, we discussed an alternative that makes RPT flows achieve fair bandwidth sharing using TCP-friendly rate control. In particular, we showed that incorporating TFRC requires careful adjustment of loss event rate, and explained how it should be done in two distinct loss

	Non-RPT	RPT(3)
TCP traffic (Mbps)	5.7	3.6
Video traffic (Mbps)	4.0	4.0
Total (Utilization)	9.7 (97%)	7.6 (76%)

Table 5: *Harm:* Bandwidth use on a RE-link.

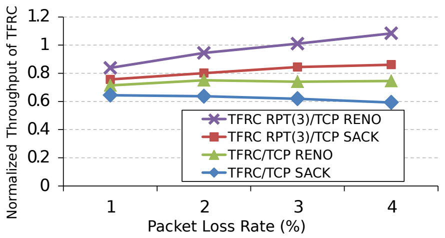


Figure 17: TFRC RPT under random loss.

patterns: *Uniform random* and *Temporal* packet loss.

Is TFRC RPT TCP-friendly? We first evaluate our scheme under the *two extreme loss patterns* created artificially, and evaluate it under a more realistic loss pattern.

Uniform random loss: In this setting, TFRC RPT behaves in a TCP friendly manner without any adjustment in the loss estimation. Figure 17 shows the normalized throughput of TFRC and TFRC RPT(3) with respect to TCP Sack and Reno under 1% to 4% random loss. TFRC RPT(3) performs slightly better than TFRC because multiple packet losses within an RTT are counted as one loss event, and therefore the loss event rate for RPT(3) is slightly lower than that of normal TFRC.

Temporal packet loss: Here, we adjust the loss event rate of TFRC RPT(r) to be r times the observed loss event rate. To validate TCP-friendliness, we evaluated the performance of TFRC RPT(3) and TFRC under the same temporal loss pattern. To create such a pattern, we generated the same cross traffic, but artificially modified the router's queue so that redundant packets do not increase the queue length. Indeed, the performance difference of the two was less than 3% with the adjusted loss event rate.

Realistic environment: The two cases appear in an inter-mixed way in practice. As discussed in §3, an adjustment factor between 1 and r is sufficient for TCP friendliness. To create realistic loss patterns, we ran TFRC with competing TCP flows. The same dumbbell topology with 1 Gbps bottleneck link capacity is used. We vary the number of competing TCP flows from 200 to 2000. Each flow's RTT is randomly selected between 40ms and 200ms. Among the TCP flows, five of them are set to have the same RTT as the TFRC flows. We compare the relative throughput of TFRC flows to the average throughput of TCP flows. Our result shows TFRC RPT(3)'s performance reasonably matches that of TCP when $\alpha = 1.5$ across various loss rates. Figure 18 shows the normalized TFRC RPT(3)'s performance with respect to TCP Reno and TCP Sack. The result show that **TFRC RPT is TCP friendly.**

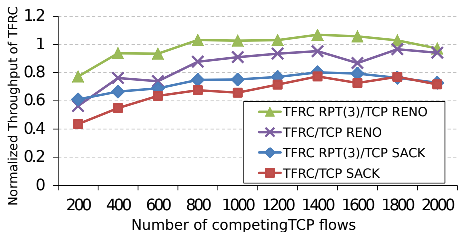


Figure 18: TFRC RPT exhibit TCP friendliness.

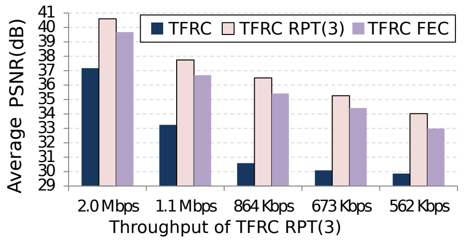


Figure 19: Video quality comparison.

Video quality: We created video streams using TFRC and TFRC RPT; in either case, we output video according to the TFRC's or TFRC RPT's sending rate.[9] We compare the video quality of normal TFRC, normal TFRC with FEC, and TFRC RPT under the same cross traffic. We vary the cross traffic to create TFRC flows whose throughputs range from 562Kbps to 2.0Mbps. For TFRC with FEC, we choose the parameter which gives the best PSNR with delay under 150ms. Figure 19 shows the video quality achieved by the TFRC flows. **We see that TFRC RPT gives the best video quality in all cases.**

7 Conclusion

This paper explores issues arising from the confluence of two trends – growing importance and volume of real-time traffic, and the growing adoption of content-aware networks. We examine a key problem at this intersection, namely that of protecting real-time traffic from data losses in content-aware networks. We show that adding redundancy in a way that network understands reduces the cost and increases the benefits of loss protection quite significantly. We refer to our candidate loss protection approach as redundant transmission (RT). Using Redundant Packet Transmission (RPT) in redundancy-elimination networks [14] as an example, we highlight various features of RT and establish that is a promising candidate to use in several practical content-aware networking scenarios. We show that RT decreases the data loss rate by orders-of-magnitude more than typical FEC schemes applicable in live video communications, and delivers higher quality video than FEC using the same bandwidth budget. RT provides fine-grained control to signal the importance of data and satisfies tight delay constraints. Yet, it is easy to use as its performance is much less sensitive to parameter selection. We show that constant bitrate RT flows

[9]For more details, refer to our technical report [31].

are prioritized over non-RT flows, but can share bandwidth fairly by incorporating TCP friendly rate control into RPT. We also show that RT provides an efficient and cost-effective loss protection mechanism in other general content-aware networks.

Acknowledgments

We thank our shepherd Xiaowei Yang and anonymous reviewers for their feedback. This research was supported in part by the National Science Foundation under awards CNS-1040757, CNS-0905134, CNS-0905277 and CNS-1040801.

References

[1] AT&T Businees Service Guide - AT&T VPN Service. `http://new.serviceguide.att.com/portals/sgportal.portal?_nfpb=true&_pageLabel=avpn_page`, 2011.

[2] Cisco Wide Area Application Services (WAAS) Software. `http://www.cisco.com/en/US/prod/collateral/contnetw/ps5680/ps6870/prod_white_paper0900aecd8051d5b2.html`, 2009.

[3] Cisco visual networking index: Forecast and methodology, 20092014. `http://www.cisco.com/`, 2010.

[4] Magic Quadrant for WAN Optimization Controllers. `http://www.gartner.com/technology/media-products/reprints/riverbed/article1/article1.html`, 2010.

[5] Juniper Networks Datasheet. `http://www.juniper.net/us/en/local/pdf/datasheets/1000113-en.pdf`, 2009.

[6] Sprint Network Performance. `https://www.sprint.net/sla_performance.php?network=pip`, 2012.

[7] YUV CIF reference videos. `http://www.tkn.tu-berlin.de/research/evalvid/cif.html`, 2010.

[8] Cisco Internal WAAS Implementation. `http://blogs.cisco.com/ciscoit/cisco_internal_waas_implementation/`, 2010.

[9] Riverbed Cloud Products. `http://www.riverbed.com/us/products/cloud_products/cloud_steelhead.php`, 2011.

[10] Riverbed Customer Stories. `http://www.riverbed.com/us/customers/index.php?filter=bandwidth`, 2011.

[11] Riverbed Steelhead Mobile. `http://www.riverbed.com/us/products/steelhead_appliance/steelhead_mobile/`, 2011.

[12] A. Albanese, J. Blöer, J. Edmonds, M. Luby, and M. Sudan. Priority encoding transmission. *IEEE Transactions on Information Theory*, 42, 1994.

[13] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data center TCP (DCTCP). In *Proc. ACM SIGCOMM*, 2010.

[14] A. Anand, A. Gupta, A. Akella, S. Seshan, and S. Shenker. Packet caches on routers: the implications of universal redundant traffic elimination. In *Proc. ACM SIGCOMM*, 2008.

[15] A. Anand, V. Sekar, and A. Akella. SmartRE: an architecture for coordinated network-wide redundancy elimination. In *Proc. ACM SIGCOMM*, 2009.

[16] G. Appenzeller, I. Keslassy, and N. McKeown. Sizing router buffers. In *Proc. ACM SIGCOMM*, 2004.

[17] M. Balakrishnan, T. Marian, K. Birman, H. Weatherspoon, and E. Vollset. Maelstrom: transparent error correction for lambda networks. In *Proc. USENIX NSDI*, 2008.

[18] A. C. Begen and Y. Altunbasak. Redundancy-controllable adaptive retransmission timeout estimation for packet video. In *Proc. ACM NOSSDAV*, 2006.

[19] J.-C. Bolot, S. Fosse-Parisis, and D. Towsley. Adaptive FEC-based error control for internet telephony. In *Proc. IEEE INFOCOM*, 1999.

[20] O. Boyaci, A. Forte, and H. Schulzrinne. Performance of video-chat applications under congestion. In *Proc. IEEE ISM*, Dec. 2009.

[21] J. W. Byers, M. Luby, M. Mitzenmacher, and A. Rege. A digital fountain approach to reliable distribution of bulk data. In *Proc. ACM SIGCOMM*, 1998.

[22] J. Cao, W. Cleveland, Y. Gao, K. Jeffay, F. Smith, and M. Weigle. Stochastic models for generating synthetic HTTP source traffic. In *Proc. IEEE INFOCOM*, volume 3, 2004.

[23] P. A. Chou, H. J. Wang, and V. N. Padmanabhan. Layered multiple description coding. In *Proc. Packet Video Workshop*, 2003.

[24] Cisco. Deploying guaranteed-bandwith services with mpls. `http://www.cisco.com/warp/public/cc/pd/iosw/prodlit/gurtb_wp.pdf`, 2012.

[25] L. De Cicco, S. Mascolo, and V. Palmisano. Skype video responsiveness to bandwidth variations. In *Proc. ACM NOSSDAV*, 2008.

[26] Emulab. Emulab. `http://www.emulab.net/`.

[27] N. Feamster and H. Balakrishnan. Packet loss recovery for streaming video. In *Proc. International Packet Video Workshop*, 2002.

[28] S. Floyd, M. Handley, J. Padhye, and J. Widmer. Equation-based congestion control for unicast applications. In *Proc. ACM SIGCOMM*, 2000.

[29] P. Frossard and O. Verscheure. Joint source/FEC rate selection for quality-optimal MPEG-2 video delivery. *IEEE Transactions on Image Processing*, 10(12), Dec. 2001.

[30] A. Hagedorn, S. Agarwal, D. Starobinski, and A. Trachtenberg. Rateless coding with feedback. In *Proc. IEEE INFOCOM*, 2009.

[31] D. Han, A. Anand, A. Akella, and S. Seshan. RPT: Re-architecting loss protection for content-aware networks. Technical Report TR-11-117, Carnegie Mellon Univ., 2011.

[32] D. Han, A. Anand, F. Dogar, B. Li, H. Lim, M. Machado, A. Mukundan, W. Wu, A. Akella, D. Andersen, J. Byers, S. Seshan, and P. Steenkiste. XIA: An architecture for an evolvable and trustworthy Internet. In *Proc. USENIX NSDI*, Apr. 2012.

[33] U. Horn, K. Stuhlmller, E. E. Herzogenrath, M. Link, and B. Girod. Robust internet video transmission based on scalable coding and unequal error protection. *Signal Processing: Image Communication*, 1999.

[34] N. Hu, L. E. Li, and Z. M. Mao. Locating Internet bottlenecks: Algorithms, measurements, and implications. In *Proc. ACM SIGCOMM*, 2004.

[35] Infineta. Velocity Dedupe Engine. `http://www.infineta.com/technology/reduce`, 2011.

[36] ITU-T. Recommendation G.114 one-way transmission time.

[37] V. Jacobson, D. K. Smetters, J. D. Thornton, M. F. Plass, N. H. Briggs, and R. L. Braynard. Networking named content. In *Proc. ACM CoNEXT*, 2009.

[38] S. Jakubczak and D. Katabi. A cross-layer design for scalable mobile video. In *Proc. ACM MobiCom*, 2011.

[39] K. Jamieson and H. Balakrishnan. PPR: Partial packet recovery for wireless networks. In *Proc. ACM SIGCOMM*, Aug. 2007.

[40] J. Klaue, B. Rathke, and A. Wolisz. EvalVid - a framework for video transmission and quality evaluation. In *Proc. Performance TOOLS*, 2003.

[41] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The click modular router. *ACM TOCS*, 2000.

[42] T. Koponen, M. Chawla, B.-G. Chun, A. Ermolinskiy, K. H. Kim, S. Shenker, and I. Stoica. A Data-Oriented (and Beyond) Network Architecture. Aug. 2007.

[43] H. Liu and M. El Zarki. Performance of H.263 video transmission over wireless channels using hybrid ARQ. *IEEE JSAC*, 15(9), Dec. 1997.

[44] D. Loguinov and H. Radha. On retransmission schemes for real-time streaming in the Internet. In *Proc. IEEE INFOCOM*, 2001.

[45] H. V. Madhyastha, T. Isdal, M. Piatek, C. Dixon, T. E. Anderson, A. Krishnamurthy, and A. Venkataramani. iPlane: An information plane for distributed services. In *Proc. 7th USENIX OSDI*, Nov. 2006.

[46] S. McCanne, M. Vetterli, and V. Jacobson. Low-complexity video coding for receiver-driven layered multicast. *IEEE JSAC*, 15(6), 1997.

[47] K. Nakauchi and K. Kobayashi. An explicit router feedback framework for high bandwidth-delay product networks. *Comput. Netw.*, 51, May 2007.

[48] L. Pantel and L. C. Wolf. On the impact of delay on real-time multiplayer games. In *Proc. NOSSDAV*, 2002.

[49] C. Papadopoulos. Retransmission-based error control for continuous media applications. In *Proc. NOSSDAV*, 1996.

[50] R. Puri and K. Ramchandran. Multiple description source coding using forward error correction codes. In *Proc. Asilomar conference on signals, systems, and computers*, 1999.

[51] I. Rhee. Error control techniques for interactive low-bit rate video transmission over the internet. In *Proc. ACM SIGCOMM*, 1998.

[52] D. Salomon. *Data Compression; the Complete Reference*. Springer, 2007.

[53] S. Sen, N. K. Madabhushi, and S. Banerjee. Scalable wifi media delivery through adaptive broadcasts. In *Proc. USENIX NSDI*, 2010.

[54] N. T. Spring and D. Wetherall. A protocol-independent technique for eliminating redundant network traffic. In *Proc. ACM SIGCOMM*, 2000.

[55] A. Suissa, J. Mellor, F. Lohier, and P. Garda. A novel video packet loss concealment algorithm & real time implementation. In *Proc. DASIP*, 2008.

[56] J. Wang and D. Katabi. ChitChat: Making video chat robust to packet loss. Technical Report MIT-CSAIL-TR-2010-031, MIT, July 2010.

[57] X. Zhu, R. Pan, N. Dukkipati, V. Subramanian, and F. Bonomi. Layered Internet video engineering (LIVE): Network-assisted bandwidth sharing and transient loss protection for scalable video streaming. In *Proc. IEEE INFOCOM*, 2010.

Serval: An End-Host Stack for Service-Centric Networking

Erik Nordström, David Shue, Prem Gopalan, Robert Kiefer
Matvey Arye, Steven Y. Ko, Jennifer Rexford, Michael J. Freedman
Princeton University

Abstract

Internet services run on multiple servers in different locations, serving clients that are often mobile and multi-homed. This does not match well with today's network stack, designed for communication between fixed hosts with topology-dependent addresses. As a result, online service providers resort to clumsy and management-intensive work-arounds—forfeiting the scalability of hierarchical addressing to support virtual server migration, directing all client traffic through dedicated load balancers, restarting connections when hosts move, and so on.

In this paper, we revisit the design of the network stack to meet the needs of online services. The centerpiece of our Serval architecture is a new Service Access Layer (SAL) that sits above an unmodified network layer, and enables applications to communicate directly on service names. The SAL provides a clean *service-level* control/data plane split, enabling policy, control, and in-stack name-based routing that connects clients to services via diverse discovery techniques. By tying *active sockets* to the control plane, applications trigger updates to service routing state upon invoking socket calls, ensuring up-to-date service resolution. With Serval, end-points can seamlessly change network addresses, migrate flows across interfaces, or establish additional flows for efficient and uninterrupted service access. Experiments with our high-performance in-kernel prototype, and several example applications, demonstrate the value of a unified networking solution for online services.

1. Introduction

The Internet is increasingly a platform for accessing services that run anywhere, from servers in the datacenter and computers at home, to the mobile phone in one's pocket and a sensor in the field. An application can run on multiple servers at different locations, and can launch at or migrate to a new machine at any time. In addition, user devices are often multi-homed (*e.g.*, WiFi and 4G) and mobile. In short, modern services operate under unprecedented *multiplicity* (in service replicas, host interfaces, and network paths) and *dynamism* (due to replica failure and recovery, service migration, and client mobility).

Yet, multiplicity and dynamism match poorly with today's host-centric TCP/IP-stack that binds connections to fixed attachment points with topology-dependent addresses and conflates service, flow, and network identifiers. This forces online services to rely on clumsy and restrictive techniques that manipulate the network layer and constrain how services are composed, managed, and controlled. For example, today's load balancers repurpose IP addresses to refer to a group of (possibly changing) service instances; unfortunately, this requires all client traffic to traverse the load balancer. Techniques for handling mobility and migration are either limited to a single layer-2 domain or introduce "triangle routing." Hosts typically cannot spread a connection over multiple interfaces or paths, and changing interfaces requires the initiation of new connections. The list goes on and on.

To address these problems, we present the Serval architecture that runs on top of an unmodified network layer. Serval provides a service-aware network stack, where applications communicate directly on *service names* instead of addresses and ports. A service name corresponds to a group of (possibly changing) processes offering the same service. This elevates services to first-class network entities (distinct from hosts or interfaces), and decouples services from network and flow identifiers. Hence, service names identify *who* one communicates with, flow names identify *what* communication context to use, while addresses tell *where* to direct the communication.

At the core of Serval is a new Service Access Layer (SAL) that sits between the transport and network layers. The SAL maps service names in packets to network addresses, based on rules in its *service table* (analogous to how the network layer uses a forwarding table). Unlike traditional "service layers," which sit *above* the transport layer, the SAL's position *below* transport provides a programmable *service-level* data plane that can adopt diverse service discovery techniques. The SAL can be programmed through a user-space control plane, acting on service-level events triggered by *active sockets* (*e.g.*, a service instance automatically registers on `bind`ing a socket). This gives network programmers hooks for ensuring service-resolution systems are up-to-date.

As such, Serval gives service providers more control over service access, and clients more flexibility in resolving services. For instance, by forwarding the first packet of a connection based on service name, the SAL can defer binding a service until the packet reaches the part of the network with fine-grain, up-to-date information. This ensures more efficient load balancing and faster failover. The rest of the traffic flows directly between end-points according to network-layer forwarding. The SAL performs signaling between end-points to establish additional flows (over different interfaces or paths) and can migrate them over time. In doing so, the SAL provides a transport-agnostic solution for interface failover, host mobility, and virtual-machine migration.

Although previous works consider some of the problems we address, none provides a comprehensive solution for service access, control, dynamicity, and multiplicity. HIP [21], DOA [30], LISP [8], LNA [5], HAIR [9] and i3 [27] decouple a host's identity from its location, but do not provide service abstractions. DONA [15] provides late binding but lacks a service-level data plane with separate control. TCP Migrate [26] supports host mobility, and MPTCP [10, 31] supports multiple paths, but both are tied to TCP and are not service-aware. Existing "backwards compatible" techniques (*e.g.*, DNS redirection, IP anycast, load balancers, VLANs, mobile IP, ARP spoofing, etc.) are point solutions suffering from poor performance or limited applicability. In contrast, Serval provides a coherent solution for service-centric networking that a simple composition of previous solutions cannot achieve.

In the next section, we rethink how the network stack should support online services, and survey related work. Then, in §3, we discuss the new abstractions offered by Serval's separation of names and roles in the network stack. Next, §4 presents our main contribution—a service-aware stack that provides a clean service-level control/data plane split. Our design draws heavily on our experiences building prototypes, as discussed in §5. Our prototype, running in the Linux kernel, already supports ten applications and offers throughput comparable to today's TCP/IP stack. In §6, we evaluate the performance of Serval-supporting replicated web services and distributed back-end storage services in datacenters. In §7, we discuss how Serval supports unmodified clients and servers for incremental deployability. The paper concludes in §8.

2. Rethinking the Network Stack

Today's stack overloads the meaning of addresses (to identify interfaces, demultiplex packets, and identify sockets) and port numbers (to demultiplex packets, differentiate service end-points, and identify application protocols). In contrast, Serval cleanly separates the roles of the *service name* (to identify a service), *flow identifiers* (to identify each flow associated with a socket), and *network addresses* (to identify each host interface). Figure 1 illustrates this comparison. Serval introduces a new Service Access Layer (SAL), above the network layer, that gives a group-based service abstraction, and shields applications and transport protocols from the multiplicity and dynamism inherent in today's online services. In this section, we discuss how today's stack makes it difficult to support online services, and review previous research on fixing individual aspects of this problem, before briefly summarizing how Serval addresses these issues.

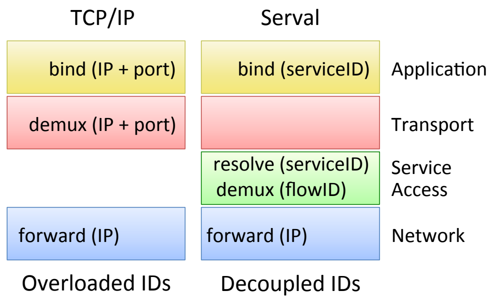


Figure 1: Identifiers and example operations on them in the TCP/IP stack versus Serval.

2.1 Application Layer

TCP/IP: Today's applications operate on two low-level identifiers (IP address and TCP/UDP port) that only implicitly name services. As such, clients must "early bind" to these identifiers using out-of-band lookup mechanisms (*e.g.*, DNS) or *a priori* knowledge (*e.g.*, Web is on port 80) before initiating communication, and servers must rely on out-of-band mechanisms to register a new service instance (*e.g.*, a DNS update protocol). Applications cache addresses instead of re-resolving service names, leading to slow failover, clumsy load balancing, and constrained mobility. A connected socket is tied to a single host interface with an address that cannot change during the socket's lifetime. Furthermore, a host cannot run multiple services with the same application-layer protocol, without exposing alternate port numbers to users (*e.g.*, "http://example.com:8080") or burying names in application headers (*e.g.*, "Host: example.com" in HTTP).

Other work: Several prior works introduce new naming layers that replace IP addresses in applications with persistent, global identifiers (*e.g.*, host/end-point identifiers [8, 21, 30], data/object names [15, 27], or service identifiers [5]), in order to simplify the handling of replicated services or mobile hosts. However, these proposals retain ports in both the stack and the API, thus not fully addressing identifier overloading. Although LNA [5], i3 [27], and DONA [15] make strong arguments for new name layers, the design of the network stack is left under-specified. A number of libraries and high-level programming languages also hide IP addresses from applications through name-based network APIs, but these merely provide programming convenience through a "traditional" application-level service layer. Such APIs do not solve the fundamental problems with identifier overloading, nor do they support late binding. Similarly, SoNS [25] can dynamically connect to services based on high-level service descriptions, but otherwise does not change the stack.

Host identifiers (as used in [5, 8, 21, 30]) still "a priori" bind to specific machines, rather than "late bind" to dynamic instances, as needed to efficiently handle churn. These host identifiers can be cached in applications (much like IP addresses), thus reducing the efficiency of load balancers. Data names in DONA [15] can late bind to hosts, but port numbers are still bound a priori. A position paper by Day *et al.* [7] argues for networking as interprocess communication, including late binding on names, but understandably does not present a detailed solution.

In Serval, **applications communicate over active sockets using service names**. Serval's *serviceIDs* offer a group abstraction that eschews host identifiers (with Serval, a host instead becomes a singleton group), enabling late binding to a service instance. Unlike application-level service layers [5], the SAL's position *below* the transport layer allows the address of a service instance to be resolved as part of connection establishment: the first packet is anycast-forwarded based on its serviceID. This further obviates the need for NAT-based load balancers that also touch the subsequent data packets. Applications automatically register with load balancers or wide-area resolution systems when they invoke active sockets, which tie application-level operations (*e.g.*, `bind` and `connect`) directly to Serval's control plane. Yet, Serval's socket API resembles existing name-based APIs that have proven popular with programmers; such a familiar abstraction makes porting existing applications easier.

2.2 Transport Layer

TCP/IP: Today's stack uses a five-tuple ⟨*remote IP, remote port, local IP, local port, protocol*⟩ to demultiplex an incoming packet to a socket. As a result, the interface addresses cannot change without disrupting ongoing connections; this is a well-known source of the TCP/IP stack's inability to support mobility without resorting to overlay indirection schemes [22, 32]. Further, today's transport layer does not support reuse of functionality [11], leading to significant duplication across different protocols. In particular, retrofitting support for migration [26] or multiple paths [31] remains a challenge that each transport protocol must undertake on its own.

Other work: Proposals like HIP [21], LNA [5], LISP [8] and DONA [15] replace addresses in the five-tuple with host or data identifiers. However, these proposals do not make any changes to the transport layer to enable reuse of functionality. TCP Migrate [26] retrofits migration support into TCP by allowing addresses in the five-tuple to change dynamically, but does not support other transport protocols. MPTCP [11, 31] extends TCP to split traffic over multiple paths, but cannot migrate the resulting flows to different addresses or interfaces. Similarly, SCTP [20] provides failover to a secondary interface, but the multi-homing support is specific to its reliable message protocol. Other recent work [11] makes a compelling case for refactoring the transport layer for a better separation of concerns (and reusable functionality), but the design does not support end-point mobility or service-centric abstractions.

In Serval, **transport protocols deal only with *data* delivery across one or more flows**, including retransmission and congestion control. Because the transport layer does not demultiplex packets, network addresses can change freely. Instead, the SAL demultiplexes packets based on ephemeral flow identifiers (*flowIDs*), which uniquely identify each flow locally on a host. By relegating the *control* of flows (*e.g.*, flow creation and migration) to the SAL, Serval allows reuse of this functionality across different transport protocols.

2.3 Network Layer

TCP/IP: Today's network layer uses hierarchical IP addressing to efficiently deliver packets. However, the hierarchical scalability of the network layer is challenged by the need for end-host mobility in a stack where upper-layer protocols fail when addresses change.

Other work: Recently, researchers and standards bodies have investigated scalable ways to support mobility while keeping network addresses fixed. This has led to numerous proposals for scalable flat addressing in enterprise and datacenter networks [2, 13, 14, 18, 19, 23]. The proposed scaling techniques, while promising, come at a cost, such as control-plane overhead to disseminate addresses [2, 23], large directory services (to map interface addresses to network attachment points) [13, 14], redirection of some data traffic over longer paths [14], network address translation to enable address aggregation [19], or continued use of spanning trees [18].

In Serval, the **network layer simply delivers packets** between end-points based on hierarchical, location-dependent addresses, just as the original design of IP envisioned. By handling flow mobility and migration *above* the network layer (*i.e.*, in the SAL), Serval allows addresses to change dynamically as hosts move.

3. Serval Abstractions

In this section, we discuss how communication on service names raises the level of abstraction in the network stack, and reduces the overloading of identifiers within and across layers.

3.1 Group-Based Service Naming

A Serval service name, called a serviceID, corresponds to a group of one or more (possibly changing) processes offering the same service. ServiceIDs are carried in network packets, as illustrated in Figure 2. This allows for service-level routing and forwarding, enables late binding, and reduces the need for deep-packet inspection in load balancers and other middleboxes. A service instance listens on a serviceID for accepting incoming connections, without exposing addresses and ports to applications. This efficiently solves issues of mobility and virtual hosting. We now discuss how serviceIDs offer considerable flexibility and extensibility in service naming.

Service granularity: Service names do not dictate the granularity of service offered by the named group of processes. A serviceID could name a single SSH daemon, a cluster of printers on a LAN, a set of peers distributing a common file, a replicated partition in a back-end storage system, or an entire distributed web service. This group abstraction hides the service granularity from clients and gives service providers control over server selection. Individual instances of a service group that must be referenced directly should use a distinct serviceID (*e.g.*, a sensor in a particular location, or the leader of a Paxos consensus group). This allows Serval to forgo host identifiers entirely, avoiding an additional name space while still making it possible to pass references to third parties. Service instances also can be assigned multiple identifiers (as in the Memcached example of §6.2, which uses hierarchical naming for partitioning with automatic failover).

Format of serviceIDs: Ultimately, system designers and operators decide what functionality to name and what structure to encode into service names. For the federated Internet, however, we imagine the need for a congruent naming scheme. For flexibility, we suggest defining a large 256-bit serviceID namespace, although other forms are possible (*e.g.*, reusing the IPv6 format could allow reuse of its existing socket API). A large serviceID namespace is attractive because a central issuing authority (*e.g.*, IANA) could allocate blocks of serviceIDs to different administrative entities, for scalable and authoritative service resolution. The block allocation ensures that a service provider can be identified by a serviceID prefix, allowing aggregation and control over service resolution. The prefix is followed by a number of bits that the delegatee can further subdivide to build service-resolution hierarchies or provide security features.

Although we advocate hierarchical service resolution for the public Internet, some services or peer-to-peer applications may use alternative, flat resolution schemes, such as those based on distributed hash tables (DHTs). In such cases, serviceIDs can be automatically constructed by combining an application-specific prefix with the hash of an application-level service (or content) name. The prefix can be omitted if the alternative resolution scheme does not coexist with other resolution schemes.

Securing communication and registration: For security, a serviceID could optionally end with a large (*e.g.*, 160-bit) self-certifying bitstring [16] that is a cryptographic hash of a service's public key and the serviceID prefix. Operating below the application layer (unlike the Web's use of SSL and certificate authorities), self-certifying serviceIDs could help move the Internet towards ubiquitous security, providing a basis for pervasive encrypted and authenticated connections between clients and servers. Self-certifying identifiers obviate the need for a single public-key infrastructure, and they turn the authentication problem into a secure bootstrapping one (*i.e.*, whether the serviceID was learned via a trusted channel).

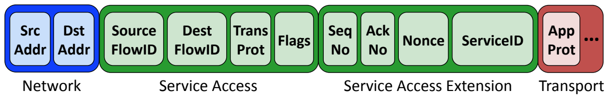


Figure 2: New Serval identifiers visible in packets, between the network and transport headers. Some additional header fields (*e.g.*, checksum, length, etc.) are omitted for readability.

Services may also seek to secure the control path that governs dynamic service registration, as otherwise an unauthorized entity could register itself as hosting the service. Even if peers authenticate one another during connection establishment, faulty registrations could serve as a denial-of-service attack. To prevent this form of attack, the registering end-point should prove that it is authorized to host the serviceID.

Serval does not dictate how serviceIDs are registered, however. For example, inside a datacenter or enterprise network, service operators may choose to secure registration through network isolation of the control channel, as opposed to cryptographic security.

When advertising service prefixes for scalable wide-area service resolution, self-certification alone is not enough to secure the advertisements. A self-certifying serviceID does not demonstrate that the originator of an advertisement is allowed to advertise a specific prefix, or that the service-level path is authorized by each intermediate hop. Such advertisements need to be secured via other means, *e.g.*, in a way similar to BGPSEC [1].

Learning service names: Serval does not dictate how serviceIDs are learned. We envision that serviceIDs are sent or copied between applications, much like URIs. We purposefully do *not* specify how to map human-readable names to serviceIDs, to avoid the legal tussle over naming [6, 29]. Users may, based on their own trust relationships, turn to directory services (*e.g.*, DNS), search engines, or social networks to resolve higher-level or human-readable names to serviceIDs, and services may advertise their serviceIDs via many such avenues.

3.2 Explicit Host-Local Flow Naming

Serval provides explicit host-local flow naming through flowIDs that are assigned and exchanged during connection setup. This allows the SAL to directly demultiplex established flows based on the destination flowID in packets, as opposed to the traditional five-tuple. Figure 2 shows the location of flowIDs in the SAL header.

Network-layer oblivious: By forgoing the traditional five-tuple, Serval can identify flows without knowing the network-layer addressing scheme. This allows Serval to transparently support both IPv4 and IPv6, without the need to expose alternative APIs for each address family.

Mobility and multiple paths: FlowIDs help identify flows across a variety of dynamic events. Such events include flows being directed to alternate interfaces or the change of an interface's address (even from IPv4 to IPv6, or vice versa), which may occur to either flow end-point. Serval can also associate multiple flows with each socket in order to stripe connections across multiple paths.

Middleboxes and NAT: FlowIDs help when interacting with middleboxes. For instance, a Serval-aware network-address translator (NAT) rewrites the local sender's network address and flowID. But because the remote destination identifies a flow solely based on its own flowID, the Serval sender can migrate between NAT'd networks (or vice versa), and the destination host can still correctly demultiplex packets.

No transport port numbers: Unlike port numbers, flowIDs do not encode the application protocol; instead, application protocols are optionally specified in transport headers. This identifier particularly aids third-party networks and service-oblivious middleboxes, such as directing HTTP traffic to transparent web caches unfamiliar with the serviceID, while avoiding on-path deep-packet inspection. Application end-points are free to elide or misrepresent this identifier, however.

Format and security: By randomizing flowIDs, a host could potentially protect against off-path attacks that try to hijack or disrupt connections. However, this requires long flowIDs (*e.g.*, 64 bits) for sufficient security, which would inflate the overhead of the SAL header. Therefore, we propose short (32-bit) flowIDs supplemented by long nonces that are exchanged only during connection setup and migrations (§4.4).

4. The Serval Network Stack

We now introduce the Serval network stack, shown in Figure 3. The stack offers a clean service-level control/data plane split: the user-space *service controller* can manage service resolution based on policies, listen for service-related events, monitor service performance, and communicate with other controllers; the Service Access Layer (SAL) provides a service-level data plane responsible for connecting to services through forwarding over *service tables*. Once connected, the SAL maps the new flow to its socket in the *flow table*, ensuring incoming packets can be demultiplexed. Using in-band signaling, additional flows can be added to a connection and connectivity can be maintained across physical mobility and virtual mi-

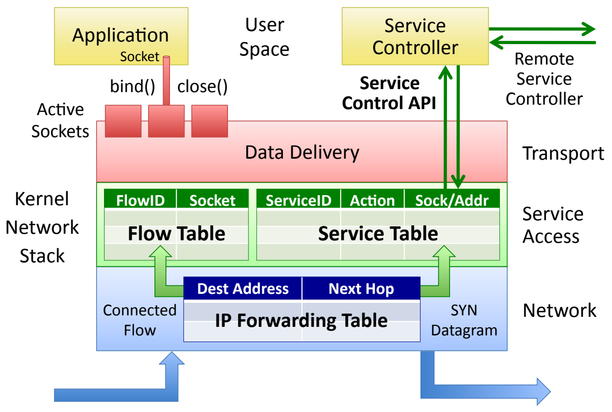


Figure 3: Serval network stack with service-level control/data plane split.

grations. Applications interact with the stack via *active sockets* that tie socket calls (*e.g.*, `bind` and `connect`) directly to service-related events in the stack. These events cause updates to data-plane state and are also passed up to the control plane (which subsequently may use them to update resolution and registration systems).

In the rest of this section, we first describe how applications interact with the stack through active sockets (§4.1), and then continue with detailing the SAL (§4.2) and how its associated control plane enables extensible service discovery (§4.3). We end the section with describing the SAL's in-band signaling protocols (§4.4).

4.1 Active Sockets

By communicating directly on serviceIDs, Serval increases the visibility into (and control over) services in the end-host stack. Through active sockets, stack events that influence service availability can be tied to a control framework that reconfigures the forwarding state, while retaining a familiar application interface.

Active sockets retain the standard BSD socket interface, and simply define a new `sockaddr` address family, as shown in Table 1. More importantly, Serval generates service-related events when applications invoke API calls. A serviceID is automatically *registered* on a call to `bind`, and *unregistered* on `close`, process termination, or timeout. Although such hooks could be added to today's network stack, they would make little sense because the stack cannot distinguish one service from another. Because servers can `bind` on serviceID prefixes, they need not `listen` on multiple sockets when they provide multiple services or serve content items named from a common prefix. While a new address family does require minimal changes to applications, porting applications is straightforward (§5.3), and a transport-level Serval translator can support unmodified applications (§7).

On a local service registration event, the stack updates the local service table and notifies the service con-

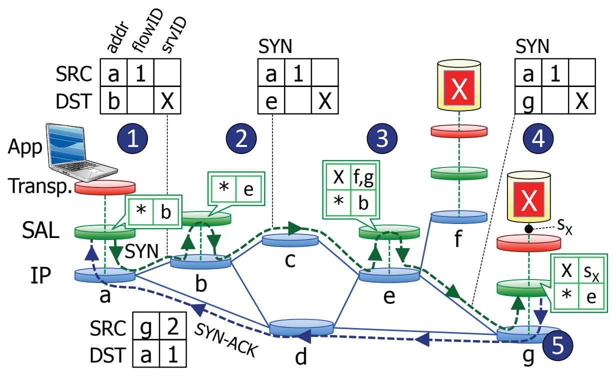


Figure 5: Establishing a Serval connection by forwarding the SYN in the SAL based on its serviceID. Client *a* seeks to communicate with service *X* on hosts *f* and *g*; devices *b* and *e* act as service routers. The default rule in service tables is shown by an "*".

again aims to access a service *X*, now available at two servers. The figure also shows service tables (present in end-points and intermediate SRs), and the SAL and network headers of the first two packets of a new connection.

When client *a* attempts to connect to service *X*, the client's SAL assigns a local flowID and random nonce to the socket, and then adds an entry in its flow table. A SYN packet is generated with the serviceID from the `connect` call, along with the new flowID and nonce. The nonce protects future SAL signaling against off-path attacks. The SAL then looks up the requested serviceID in its service table, but with no local listening service (DEMUX rule) or known destination for *X*, the request is sent to the IP address *b* of the default FORWARD rule (as shown by the figure's Step 1).

Finding no more specific matches, SR *b* again matches on its default FORWARD rule, and directs the packet to the next hop SR *e* (Step 2) by rewriting the IP destination in the packet.[1] This forwarding continues recursively through *e* (Step 3), until reaching a listening service end-point s_x on host *g* (Step 4), which then creates a responding socket with a new flowID, also updating its flow table. End-host *g*'s SYN-ACK response (Step 5) includes its own address, flowID, and nonce. The SYN-ACK and all subsequent traffic in both directions travel directly between the end-points, bypassing SRs *b* and *e*. After the first packet, all remaining packets can be demultiplexed by the flow table based on destination flowIDs, without requiring the SAL extension header.

The indirection of the SYN may increase its delay, although data packets are unaffected. In comparison, the lack of indirection support in today's stack requires putting load balancers on data paths, or tunneling all packets from the one location to another when hosts move.

[1]The SR also may rewrite the source IP (saving the original in a SAL extension header) to comply with ingress filtering.

4.3 A Service Control Plane for Extensible Service Discovery

To handle a wide range of services and deployment scenarios, Serval supports diverse ways to register and resolve services. The service-level control/data plane split is central to this ability; the controller disseminates serviceID prefixes to build service resolution networks, while the SAL applies rules to packets, sending them onward—if necessary, through service routers deeper in the network—to a remote service instance. The SAL does not control *which* forwarding rules are in the service table, *when* they are installed, or *how* they propagate to other hosts. Instead, the local service controller (i) manages the state in the service table and (ii) potentially propagates it to other service controllers. Depending on which rules the controller installs, when it installs them (reactively or proactively), and what technique it uses to propagate them, Serval can support different deployment scenarios.

Wide-area service routing and resolution. Service prefix dissemination can be performed similarly to existing inter/intra-domain routing protocols (much like LISP-ALT uses BGP [12]). A server's controller can "announce" a new service instance to an upstream service controller that, in turn, disseminates reachability information to a larger network of controllers. This approach enables enterprise-level or even wide-area service resolution. Correspondingly, serviceIDs can be aggregated by administrative entities for scalability and resolution control. For example, a large organization like Google could announce coarse-grained prefixes for top-level services like Search, Gmail, or Documents, and only further refine its service naming within its backbone and datacenters. This prefix allocation gives organizations control over their authoritative service resolvers, reduces resolution stretch, and minimizes churn. On the client, the service table would have FORWARD rules to direct a SYN packet to its local service router, which in turn directs the request up the service router hierarchy to reach a service instance.

Peer-to-peer service resolution: As mentioned in §3.1, peer-to-peer applications may resolve through alternative resolution networks, such as DHT-based ones. In such cases, a hash-based serviceID is forwarded through service tables, ultimately registering or resolving with a node responsible for the serviceID. This DHT-based resolution can coexist with an IANA-controlled resolution hierarchy, however, as both simply map to different rules in the same service table. Yet, DHTs generally limit control over service routers' serviceID responsibilities and increase routing stretch, so are less appropriate as the primary resolution mechanism for the federated Internet.

Ad hoc service access: Without infrastructure, Serval can perform service discovery via broadcast flooding. Using a "default" rule, the stack broadcasts a service request

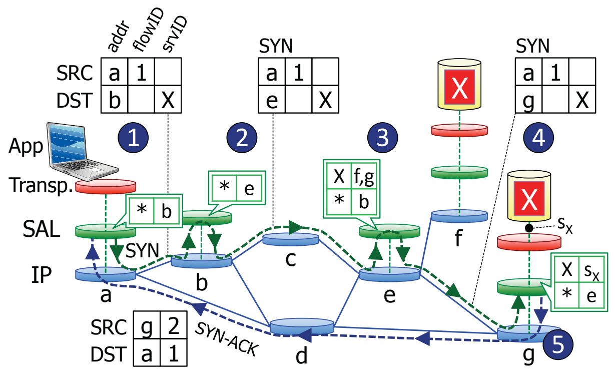


Figure 5: Establishing a Serval connection by forwarding the SYN in the SAL based on its serviceID. Client *a* seeks to communicate with service *X* on hosts *f* and *g*; devices *b* and *e* act as service routers. The default rule in service tables is shown by an "*".

again aims to access a service *X*, now available at two servers. The figure also shows service tables (present in end-points and intermediate SRs), and the SAL and network headers of the first two packets of a new connection.

When client *a* attempts to connect to service *X*, the client's SAL assigns a local flowID and random nonce to the socket, and then adds an entry in its flow table. A SYN packet is generated with the serviceID from the `connect` call, along with the new flowID and nonce. The nonce protects future SAL signaling against off-path attacks. The SAL then looks up the requested serviceID in its service table, but with no local listening service (DEMUX rule) or known destination for *X*, the request is sent to the IP address *b* of the default FORWARD rule (as shown by the figure's Step 1).

Finding no more specific matches, SR *b* again matches on its default FORWARD rule, and directs the packet to the next hop SR *e* (Step 2) by rewriting the IP destination in the packet.[1] This forwarding continues recursively through *e* (Step 3), until reaching a listening service endpoint s_x on host *g* (Step 4), which then creates a responding socket with a new flowID, also updating its flow table. End-host *g*'s SYN-ACK response (Step 5) includes its own address, flowID, and nonce. The SYN-ACK and all subsequent traffic in both directions travel directly between the end-points, bypassing SRs *b* and *e*. After the first packet, all remaining packets can be demultiplexed by the flow table based on destination flowIDs, without requiring the SAL extension header.

The indirection of the SYN may increase its delay, although data packets are unaffected. In comparison, the lack of indirection support in today's stack requires putting load balancers on data paths, or tunneling all packets from the one location to another when hosts move.

[1]The SR also may rewrite the source IP (saving the original in a SAL extension header) to comply with ingress filtering.

4.3 A Service Control Plane for Extensible Service Discovery

To handle a wide range of services and deployment scenarios, Serval supports diverse ways to register and resolve services. The service-level control/data plane split is central to this ability; the controller disseminates serviceID prefixes to build service resolution networks, while the SAL applies rules to packets, sending them onward—if necessary, through service routers deeper in the network—to a remote service instance. The SAL does not control *which* forwarding rules are in the service table, *when* they are installed, or *how* they propagate to other hosts. Instead, the local service controller (i) manages the state in the service table and (ii) potentially propagates it to other service controllers. Depending on which rules the controller installs, when it installs them (reactively or proactively), and what technique it uses to propagate them, Serval can support different deployment scenarios.

Wide-area service routing and resolution. Service prefix dissemination can be performed similarly to existing inter/intra-domain routing protocols (much like LISP-ALT uses BGP [12]). A server's controller can "announce" a new service instance to an upstream service controller that, in turn, disseminates reachability information to a larger network of controllers. This approach enables enterprise-level or even wide-area service resolution. Correspondingly, serviceIDs can be aggregated by administrative entities for scalability and resolution control. For example, a large organization like Google could announce coarse-grained prefixes for top-level services like Search, Gmail, or Documents, and only further refine its service naming within its backbone and datacenters. This prefix allocation gives organizations control over their authoritative service resolvers, reduces resolution stretch, and minimizes churn. On the client, the service table would have FORWARD rules to direct a SYN packet to its local service router, which in turn directs the request up the service router hierarchy to reach a service instance.

Peer-to-peer service resolution: As mentioned in §3.1, peer-to-peer applications may resolve through alternative resolution networks, such as DHT-based ones. In such cases, a hash-based serviceID is forwarded through service tables, ultimately registering or resolving with a node responsible for the serviceID. This DHT-based resolution can coexist with an IANA-controlled resolution hierarchy, however, as both simply map to different rules in the same service table. Yet, DHTs generally limit control over service routers' serviceID responsibilities and increase routing stretch, so are less appropriate as the primary resolution mechanism for the federated Internet.

Ad hoc service access: Without infrastructure, Serval can perform service discovery via broadcast flooding. Using a "default" rule, the stack broadcasts a service request

(SYN) and awaits a response from (at least) one service instance. Any listening instances on the local segment may respond, and the client can select one from the responses (typically the first). On the server side, on a local registration event, the controller can either (i) be satisfied with the DEMUX rule installed locally (which causes the SAL to listen for future requests) or (ii) flood the new mapping to other controllers (causing them to install FORWARD rules and thus prepopulate the service tables of prospective clients). Similarly, on an unregistration event, (i) the local DEMUX rule is deleted and (ii) the controller can flood a message to instruct others to delete their specific FORWARD mapping. Ad hoc mode can operate without a name-resolution infrastructure (at the cost of flooding), and can also be used for bootstrapping (*i.e.*, to discover a service router). It also extends to multihop ad hoc routing protocols, such as OLSR or DYMO; flooding a request/solicitation for a (well-known) serviceID makes more sense than using an address, since ad hoc nodes typically do not know the address of a service.

Lookup with name-resolution servers: A controller can also install service table rules "on demand" by leveraging directory services. A controller installs a DELAY rule (either a default "catch-all" rule or one covering a certain prefix), and matching packets are buffered while the controller resolves their serviceIDs. This design allows the controller to adopt different query/response protocols for resolution, including legacy DNS. A returned mapping is installed as a FORWARD rule and the controller signals the stack to re-match the delayed packets. The resolution can similarly be performed by an *in-network* lookup server; the client's service table may FORWARD the SYN to the lookup server, which itself DELAYs, resolves, and subsequently FORWARDs the packet towards the service destination. Upon registration or unregistration events, a service controller sends update messages to the lookup system, similar to dynamic DNS updates [28].

In addition to these high-level approaches, various hybrid solutions are possible. For instance, a host could broadcast to reach a local service router, which may hierarchically route to a network egress, which in turn can perform a lookup to identify a remote datacenter service router. This authoritative service router can then direct the SYN packet to a particular service instance or subnetwork. These mechanisms can coexist simultaneously—much like the flexibility afforded by today's inter- and intra-domain routing protocols—they simply are different service rules that are installed when (and where) appropriate for a given scenario.

4.4 End-Host Signaling for Multiple Flows and Migration

To support multiplicity and dynamism, the SAL can establish multiple flows (over different interfaces or paths) to a remote end-point, and seamlessly migrate flows over time. The SAL's signaling protocols are similar to MPTCP [10, 31] and TCP Migrate [26], with some high-level differences. First, control messages (*e.g.*, for creating and tearing down flows) are separate from the data stream with their own sequence numbers. Second, by managing flows in a separate layer, Serval can support transport protocols other than TCP. Third, our solution supports both multiple flows *and* migration.

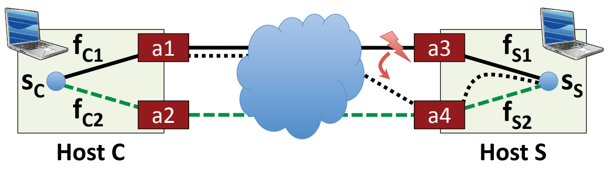


Figure 6: Schematic showing relationship between sockets, flowIDs, interfaces, addresses, and paths.

Multi-homing and multi-pathing: Serval can split a socket's data stream across multiple flows established and maintained by the SAL on different paths. Consider the example in Figure 6, where two multi-homed hosts have a socket that consists of two flows. The first flow, created when the client C first connects to the server S, uses local interface $a1$ and flowID f_{C1} (and interface $a3$ and flowID f_{S1} on S). On any packet, either host can piggyback a list of other available interfaces (*e.g.*, $a2$ for C, and $a4$ for S) in a SAL extension header, to enable the other host to create additional flows using a similar three-way handshake. For example, if S's SYN-ACK packet piggybacks information about interface address $a4$, C could initiate a second flow from $a2$ to $a4$.

Connection affinity across migration: Since the transport layer is unaware of flow identifiers and interface addresses, the SAL can freely migrate a flow from one address, interface, or path to another. This allows Serval to support client mobility, interface failover, and virtual machine migration with a single simple flow-resynchronization primitive. Obviously, these changes would affect the round-trip time and available bandwidth between the two end-points, which, in turn, affect congestion control. Yet, this is no different to TCP than any other sudden change in path properties. Further, the SAL can notify transport protocols on migration events to ensure quick recovery, *e.g.*, by temporarily freezing timers.

Returning to Figure 6, suppose the interface with address $a3$ at server S fails. Then, then server's stack can move the ongoing flow to another interface (*e.g.*, the interface with address $a4$). To migrate the flow, S sends C an RSYN packet (for "resynchronize") with flowIDs $\langle f_{S1}, f_{C1} \rangle$ and the new address $a4$. The client returns an RSYN-ACK, while waiting for a final acknowledgment to confirm the change. Sequence numbers in the resynchronization messages ensure that the remote end-points track changes in the identifiers correctly across multiple changes, even if RSYN and RSYN-ACK messages arrive out of order. To ensure correctness, we formally verified our resynchronization protocol using the Promela language and SPIN verification tool [4].

In the rare case that both end-points move at the same time, neither end-point would receive the other's RSYN packet. To handle simultaneous migration, we envision directing an RSYN through a mobile end-point's old service router to reestablish communication. Similar to Mobile IP [22], the service router acts as a "home agent," but only *temporarily* to ensure successful resynchronization.

The signaling protocol has good security and backwards-compatibility properties. Random flow nonces protect against off-path attacks that try to hijack or disrupt connections. Off-path attackers would have to brute-force guess these nonces, which is impractical. This solution does not mitigate on-path attacks, but this is no less secure than existing, non-cryptographic protocols. The signaling protocol can also operate correctly behind NATs. Much like legacy NATs can translate ports, a Serval NAT translates both flowIDs and addresses. Optional UDP encapsulation also ensures operation behind legacy NATs.

5. Serval Prototype

An architecture like Serval would be incomplete without implementation insights. Our prototyping effort was instrumental in refining the design, leading to numerous revisions as our implementation matured. Through prototyping, we have (i) learned valuable lessons about our design, its performance, and scalability, (ii) explored incremental-deployment strategies, and (iii) ported applications to study how Serval abstractions benefit them. In this section, we describe our Serval prototype and expand on these three aspects.

5.1 Lessons From the Serval Prototype

Our Serval stack consists of about 28,000 lines of C code, excluding support libraries, test applications, and daemons. The stack runs natively in the Linux kernel as a module, which can be loaded into an unmodified and running kernel. The module also runs on Android, enabling mobile devices to migrate connections between WiFi and cellular interfaces. An abstraction layer allows the stack to optionally run as a user-space daemon on top of raw IP sockets. This allows the stack to run on other platforms (such as BSD) and to be deployed on testbeds (like PlanetLab) that do not allow kernel modules. The user-mode capability of the stack also helps with debugging.

The prototype supports most features—migration, SAL forwarding, etc.—with the notable exception of multipath, which we plan to add in the future. The SAL implements the service table (with FORWARD and DEMUX rules), service resolution, and end-point signaling. The service controller interacts with the stack via a Netlink socket, installing service table rules and reacting on socket calls (`bind`, `connect`, etc.). The stack supports TCP and UDP equivalents, where UDP can operate in both connected mode (with service instance affinity) and unconnected mode (with every packet routed through the service table).

Interestingly, our initial design did not have a full SAL and service table, and instead implemented much of the service controller functionality directly in the stack (*e.g.*, sending (un)registration messages). The stack forwarded the first packet of each connection to a "default" service router (like a default gateway). However, this design led to two distinct entities with different functionality: the end-host and the service router, leaving end-hosts with little control and flexibility. For example, hosts could not communicate "ad hoc" on the same segment without a service router, and could not adopt other service-discovery techniques. The service router, which implemented most of its functionality in user space, also had obvious performance issues—especially when dealing with unconnected datagrams that all pass through the service router. This made us realize the need for a clean service-level control/data plane split that could cater to both end-hosts and routers. In fact, including the SAL, service table, and control interface in our design allowed us to unify the implementations of service routers and end-hosts, with only the policy defining their distinct roles.

The presence of a service table also simplified the handling of "listening" sockets, as it eventually evolved into a general rule-matching table, which allows demultiplexing to sockets as well as forwarding. The ability to demultiplex packets to sockets using LPM enables new types of services (*e.g.*, ones that serve content sharing one prefix).

The introduction of the SAL inevitably had implications for the transport layer, as a goal was to be able to late bind connections to services. Although we could have modified each transport protocol separately, providing a standard solution in the SAL made more sense. Further, since today's transport protocols need to read network-layer addresses for demultiplexing purposes, changes are necessary to fully support migration and mobility. Another limitation of today's transport layer is the limited signaling they allow. TCP extensions (*e.g.*, MPTCP and TCP Migrate) typically implement their signaling protocols in TCP options for compatibility reasons. However, these options are protocol specific, can only be piggybacked on packets in the data stream, and cannot consume sequence space themselves. Options are also unreliable, since they can be stripped or packets resegmented by middleboxes. To side-step these difficulties, the SAL uses its own sequence numbers for control messages.

We were presented with two approaches for rewiring the stack: using UDP as a base for the SAL (as advocated in [11]), or using our own "layer-3.5" protocol headers.

TCP	Mean	Stdev
Stack	Mbit/s	Mbit/s
TCP/IP	934.5	2.6
Serval	933.8	0.03
Translator	932.1	1.5

UDP	Tput	Pkts	Loss
Router	Mbit/s	Kpkt/s	Loss %
IP Forwarding	957	388.4	0.79
Serval	872	142.8	0.40

Table 2: TCP throughput of the native TCP/IP stack, the Serval stack, and the two stacks connected through a translator. UDP routing throughput of native IP forwarding and the Serval stack.

The former approach would make our changes more transparent to middleboxes and allow reuse of an established header format (*e.g.*, port fields would hold flowIDs). However, this solution requires "tricks" to be able to demultiplex both legacy UDP packets and SAL packets when both look the same. Defining our own SAL headers therefore presented us with a cleaner approach, with optional UDP encapsulation for traversing legacy NATs and other middleboxes. Recording addresses in a SAL extension headers also helps comply with ingress filtering (§7).

In Serval, the transport layer does not perform connection establishment, management, and demultiplexing. Despite this seemingly radical change, we could adapt the Linux TCP code with few changes. Serval only uses the TCP functionality that corresponds to the ESTABLISHED state, which fortunately is mostly independent from the connection handling. In the Serval stack, packets in an established data stream are simply passed up from the SAL to a largely unmodified transport layer. If anything, transport protocols are *less* complex in Serval, by having shared connection logic in the SAL. Our stack coexists with the standard TCP/IP stack, which can be accessed simultaneously via `PF_INET` sockets.

5.2 Performance Microbenchmarks

The first part of Table 2 compares the TCP performance of our Serval prototype to the regular Linux TCP/IP stack. The numbers reflect the average of ten 10-second TCP transfers using `iperf` between two nodes, each with two 2.4 GHz Intel E5620 quad-core CPUs and GigE interfaces, running Ubuntu 11.04. Serval TCP is very close to regular TCP performance and the difference is likely explained by our implementation's lack of some optimizations. For instance, we do not support hardware checksumming and segmentation offloading due to the new SAL headers. Furthermore, we omitted several features, such as SACK, FACK, DSACK, and timestamps, to simplify the porting of TCP. We plan to add these features in the future. We speculate that the lack of optimizations may also explain Serval's lower stdev, since the system does not drive the link to very high utilization, where even modest variations in cross traffic would lead to packet loss and delay. The table also includes numbers for our translator (§7), which allows legacy hosts to communicate with Serval hosts. The translator (in this case running on a third intermediate host) uses Linux's `splice` system call to zero-copy data between a legacy TCP socket and a Serval TCP socket, achieving high performance. As such, the overhead of translation is minimal.

Application	Vers.	Codebase	Changes
Iperf	2.0.0	5,934	240
TFTP	5.0	3,452	90
PowerDNS	2.9.17	36,225	160
Wget	1.12	87,164	207
Elinks browser	0.11.7	115,224	234
Firefox browser	3.6.9	4,615,324	70
Mongoose webserver	2.10	8,831	425
Memcached server	1.4.5	8,329	159
Memcached client	0.40	12,503	184
Apache Bench / APR	1.4.2	55,609	244

Table 3: Applications currently ported to Serval.

The second part of Table 2 depicts the relative performance of a Serval service router versus native IP forwarding. Here, two hosts run `iperf` in unconnected UDP mode, with all packets forwarded over an intermediate host through either the SAL or just plain IP. Throughput was measured using full MSS packets, while packet rate was tested with 48-byte payloads (equating to a Serval SYN header) to represent resolution throughput. Serval achieves decent throughput (91% of IP), but suffers significant degradation in its packet rate due to the overhead of its service table lookups. Our current implementation uses a bitwise trie structure for LPM. With further optimizations, like a full level-compressed trie and caching (or even TCAMs in dedicated service routers), we expect to bridge the performance gap considerably.

5.3 Application Portability

We have added Serval support to a range of network applications to demonstrate the ease of adoption. Modifications typically involve adding support for a new `sockaddr_sv` socket address to be passed to BSD socket calls. Most applications already have abstractions for multiple address types (*e.g.*, IPv4/v6), which makes adding another one straightforward.

Table 3 overviews the applications we have ported and the lines of code changed. Running the stack in user-space mode necessitates renaming API functions (*e.g.*, `bind` becomes `bind_sv`). Therefore, our modifications are larger than strictly necessary for kernel-only operation. In our experience, adding Serval support typically takes a few hours to a day, depending on application complexity.

6. Experimental Case Studies

To demonstrate how Serval enables diverse services, we built several example systems that illustrate its use in managing a large, multi-tier web service. A typical configuration of such a service places customer-facing webservers—all offering identical functionality—in a datacenter. Using

Serval, clients would identify the entire web service by a single serviceID (instead of a single IP address per site or load balancer, for example).

The front-end servers typically store durable customer state in a partitioned back-end distributed storage system. Each partition handles only a subset of the data, and the webservers find the appropriate storage server using a static and manually configured mapping (as in the Memcached system [17]). Using Serval, this mapping can be made dynamic, and partitions redistributed as storage servers are added, removed, or fail.

Other forms of load balancing can also achieve higher performance or resource utilization. Today's commodity servers typically have 2–4 network interfaces; balancing traffic across interfaces can lead to higher server throughput and lower path-level congestion in the datacenter network. Yet, connections are traditionally fixed to an interface once established; using Serval, this mapping can be made dynamic and driven either by local measurements or externally by a centralized controller [3].

In a cloud setting, webservers may run in virtual machines that can be migrated between hosts to distribute load. This is particularly attractive to "public" cloud providers, such as Amazon (EC2) or Rackspace (Mosso), which do not have visibility into or control over service internals. Traditionally, however, VMs can be migrated only within a layer-2 subnet, of which large datacenters have many, since network connections are bound to fixed IP addresses and migration relies on ARP tricks.

The section is organized around example systems we built for each of these tasks: a replicated front-end web service that provides dynamic load balancing between servers (§6.1), a back-end storage system that uses partitioning for scalability (§6.2), and servers that migrate individual connections between interfaces or entire virtual machines, to achieve higher utilization (§6.3).

6.1 Replicated Web Services

To demonstrate Serval's use for dynamic service scaling through anycast service resolution, we ran an experiment representative of a front-end web cluster. Four clients running the Apache benchmark generate requests to a Serval web service with an evolving set of Mongoose service instances. For load balancing, a single service router receives service updates and resolves requests. As in §5.2, all hosts are connected to the same ToR switch via GigE links. To illustrate the load-balancing effect on throughput and request rate, each client requests a 3MB file and maintains an open window of 20 HTTP requests, which is enough demand to fully saturate their GigE link.

Figure 7 shows the total throughput and request rate achieved by the Serval web service. Initially, from time 0 to 60 seconds, two Mongoose webserver instances serve a total of 80 req/s, peaking around 1800 Mbps, effectively

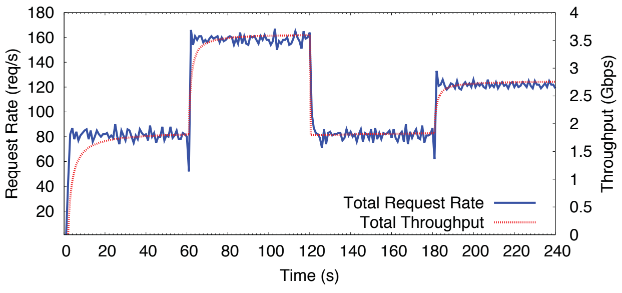


Figure 7: Total request rate and throughput for a replicated web service as servers join and leave (every 60 seconds). Request rate and throughput are proportional to the number of active service instances, with each server saturating its 1 GigE link.

saturating the server bandwidth. At time 60, two new service instances start, register with the service router—simply by `bind`ing to the appropriate serviceID, as the stack and local controller take care of the rest—and immediately begin to serve new requests. The total system request rate at this point reaches 160 req/s and 3600 Mbps. At time 120, we force the first two servers to gracefully shut down, which causes them to `close` their listening socket (and hence unregister with the service router), finish ongoing transfers, then exit. The total request rate and throughput drops back to the original levels without further degradation in service. Finally, we start another server at time 180, which elevates the system request rate to 120 req/s and 2700 Mbps.

Serval is able to maintain a request rate and throughput proportional to the number of servers, without an expensive, dedicated load balancer. Moreover, Serval distributes client load evenly across each instance, allowing the system to reach full saturation.

By acting on service registration and unregistration events generated by the Serval stack, the service router can respond instantly to changes in service capacity and availability. An application-level resolution service (*e.g.*, DNS, LDAP, etc) would require additional machinery to monitor service liveness and have to contend with either the extra RTT of an early-binding resolution or stale caches. Alternatively, using an on-path layer-7 switch or VIP/DIP load balancer would require aggregate bandwidth commensurate to the number of clients and servers (8 Gbps in this case). A half-NAT solution would remove the bottleneck for response traffic, which would be highly effective for web (HTTP GET) workloads. However, it still constrains incoming request traffic, which can hinder cloud services with high bidirectional traffic (*e.g.*, Dropbox backup or online gaming).

In contrast, the service router is simply another node on the rack with the same GigE interface to the top-of-rack switch. After all, it is only involved with connection establishment, not actual data transfer. Although each server has a unique IP in this scenario, even in the case where servers share a virtual IP (either to conserve address space or mask datacenter size), Serval simplifies the task of NAT boxes by offloading the burden of load balancing and server monitoring to the service routers.

6.2 Back-End Distributed Storage

To illustrate Serval's use in a partitioned storage system, we implemented a *dynamic* Memcached system. Memcached provides a simple key-value GET/SET caching service, where "keyspace" partitions are spread over a number of servers for load balancing. Clients map keys to partitions using a static resolution algorithm (*e.g.*, consistent hashing), and send their request to a server according to a *static* list that maps partitions to a corresponding server. However, this static mapping complicates repartitioning when servers are added, removed, or fail.

With Serval, the partition mapping can be made dynamic, by allowing clients to issue request directly to a serviceID constructed from a "common" Memcached prefix, followed by the content key. The SAL then maps the serviceID to a partition using LPM in the service table, and ultimately forwards the request to a responsible server that listens on the common prefix. Response packets travel directly to the client, bypassing the service router. A potential downside of this SAL forwarding is that clients cannot easily aggregate requests on a per-server basis, having no knowledge of partition assignments. Instead, aggregation could be handled by a service router, at the cost of increased latency.

When Memcached servers register and unregister with the network (or are overloaded), the control plane reassigns partition(s) by simply changing rules in service tables. For reliability and ease of management, service tables can cover several partitions by a single prefix, giving the option of having "fallback" rules when more specific rules are evicted from the table (*e.g.*, due to failures). This reduces both the strain on the registration system and the number of cache misses during partition changes.

It is common that clients use TCP for SETs (for reliability and requests larger than one datagram) and UDP for GETs (for reduced delay and higher throughput). SAL forwarding makes more sense in combination with UDP, however, as TCP uses a persistent connection per server and thus still requires management of these connections by the application. Reliability can be implemented on top of UDP with a simple acknowledgment/retry scheme.[2]

Figure 8 illustrates the behavior of Memcached on Serval using one client, four servers, and an intermediate service router. The service router and client run on the same spec machines as our microbenchmarks (§5.2), while the Memcached servers run on machines with two

[2]In fact, many large-scale services avoid the overhead of TCP by implementing application-level flow control for UDP.

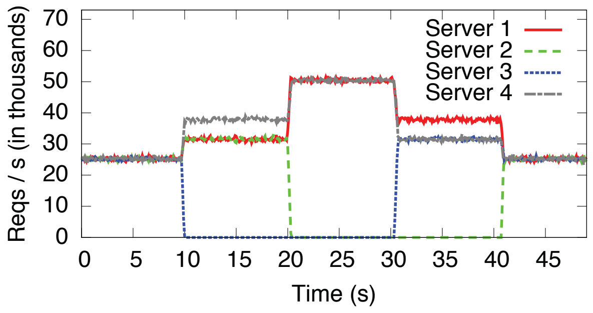


Figure 8: As Memcached instances join or leave (every 10 seconds in the experiment), Serval transparently redistributes the data partitions over the available servers.

2.4 GHz AMD Opteron 2376 quad-core CPUs, also with GigE interfaces. The service router assigns each server four partitions (*i.e.*, it uses the last 4-bits of the serviceID prefix to assign a total of 16 partitions) and reassigns them as servers join or leave. The client issues SET requests (each with a data object of 1024 bytes) with random keys at a rate of 100,000 requests per second. In the beginning, all four Memcached servers are operating. Around the 10-second mark, one server is removed, and the service router distributes the server's four partitions among the three remaining servers (giving two partitions to one of them, as visible in the graph). Another server is removed at the 20-second mark, evenly distributing the partitions (and load) on the remaining two servers. The two failed servers join the cluster again at the 30-second and 40-second marks, respectively, offloading partitions from the other servers. Although simple, this experiment effectively shows the dynamicity that back-end services can support with Serval. Naturally, more elaborate hierarchical prefix schemes can be devised, in combination with distributed service table states, to scale services further.

6.3 Interface Load Balancing and Virtual Machine Migration

Modern commodity servers have multiple physical interfaces. With Serval, a server can `accept` a connection on one interface, and then migrate it to a different interface (possibly on a different layer-3 subnet) without breaking connectivity. To demonstrate this functionality, we ran an `iperf` server on a host with two GigE interfaces. Two `iperf` clients then `connect`ed to the server and began transfers to measure maximum throughput, as shown in Figure 9. Given TCP's congestion control, each connection achieves a throughput of approximately 500 Mbps when connected to the same server interface. Six seconds into the experiment, the server's service controller signals the SAL to migrate one flow to its second interface. TCP's congestion control adapts to this change in capacity, and

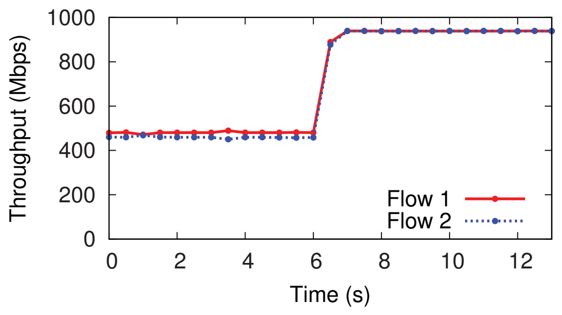


Figure 9: A Serval server migrates one of the flows sharing a GigE interface to a second interface, yielding higher throughput for both.

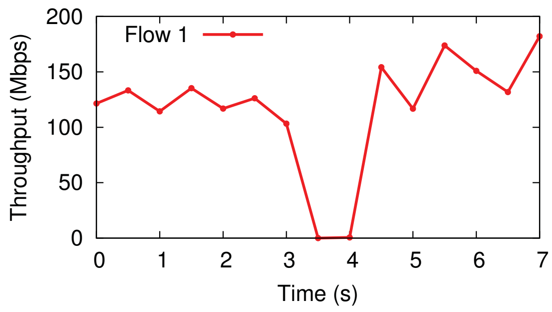


Figure 10: A VM migrates across subnets, causing a short interruption in the data flow.

both connections quickly rise to their full link capacity approaching 1 Gbps.

Cloud providers can also use Serval's migration capabilities to migrate virtual machines across layer-3 domains. Figure 10 illustrates such a live VM migration that maintains a data flow across a migration from one physical host to another, each on a different subnet. After the VM migration completes, TCP stalls for a short period, during which the VM is assigned a new address and performs an RSYN handshake.

7. Incremental Deployment

This section discusses how Serval can be used by unmodified clients and servers through the use of TCP-to-Serval (or Serval-to-TCP) *translators*. While Section 4.3 discussed backwards-compatible approaches for simplifying network infrastructure deployment (*e.g.*, by leveraging DNS), we now address supporting unmodified applications and/or end-hosts. For both, the application uses a standard `PF_INET` socket, and we map legacy IP addresses and ports to serviceIDs and flowIDs.

Supporting unmodified applications: If the end-host installs a Serval stack, translation between legacy and Serval packets can be done on-the-fly without terminating a connection: A virtual network interface can capture legacy packets to particular address blocks, then translate the legacy IP addresses and ports to Serval identifiers.

Supporting unmodified end-hosts: A TCP-to-Serval translator can translate legacy connections from unmodified end-hosts to Serval connections. To accomplish this on the client-side, the translator needs to (i) know which service a client desires to access and (ii) receive the packets of all associated flows. Several different deployment scenarios can be supported.

To deploy this translator as a client-side middlebox, one approach has the client use domain names for service names, which the translator will then transparently map to a private IP address, as a surrogate for the serviceID. In particular, to address (i), the translator inserts itself as a recursive DNS resolver in between the client and an upstream resolver (by static configuration in `/etc/resolv.conf` or by DHCP). Non-Serval-related DNS queries and replies are handled as normal. If a DNS response holds a Serval record, however, the serviceID and FORWARD rule are cached in a table alongside a new private IP address. The translator allocates this private address as a local traffic sink for (ii)—hence subsequently responding to ARP requests for it—and returns it to the client as an A record.

Alternatively, large service providers like Google or Yahoo!, spanning many datacenters, could deploy translators in their many Points-of-Presence (PoP). This would place service-side translators nearer to clients—similar to the practice of deploying TCP normalization and HTTP caching. The translators could identify each of the provider's services with a unique public IP:port. The client could resolve the appropriate public IP address (and thus translator) through DNS.

As mentioned in §5.2, we implemented such a service-side TCP-to-Serval translator [24]. When receiving a new client connection, the translator looks up the appropriate serviceID, and initiates a new Serval connection. It then transfers data back-and-forth between each socket, much like a TCP proxy. As shown in our benchmarks, the translator has very little overhead.

A Serval-to-TCP/UDP translator for unmodified servers looks similar, where the translator converts a Serval connection into a legacy transport connection with the server's legacy stack. A separate liveness monitor can poll the server for service (un)registration events.

In fact, both translators can be employed simultaneously, *e.g.*, to allow smartphones to transparently migrate the connections of legacy applications between cellular and WiFi networks. On an Android device, `iptables` rules can direct the traffic to any specified TCP port to a locally-running TCP-to-Serval translator, which connects to a remote Serval-to-TCP translator,[3] which in turn communicates with the original, unmodified destination.

[3] In our current implementation of such two-sided proxying, the client's destination is inserted at the beginning of the Serval TCP stream and parsed by the remote translator.

Handling legacy middleboxes: Legacy middleboxes can drop packets with headers they do not recognize, thus frustrating the deployment of Serval. To conform to middlebox processing, Serval encapsulates SAL headers in shim UDP headers, as described in §5. The SAL records the addresses of traversed hosts in a "source" extension of the first packet, allowing subsequent (response) packets to traverse middleboxes in the reverse order, if necessary.

8. Conclusions

Accessing diverse services—whether large-scale, distributed, ad hoc, or mobile—is a hallmark of today's Internet. Yet, today's network stack and layering model still retain the static, host-centric abstractions of the early Internet. This paper presents a new end-host stack and layering model, and the larger Serval architecture for service discovery, that provides the right abstractions and protocols to more naturally support service-centric networking. We believe that Serval is a promising approach that makes services easier to deploy and scale, more robust to churn, and more adaptable to diverse deployment scenarios. More information and source code are available at `www.serval-arch.org`.

Acknowledgments. We thank David Andersen, Laura Marie Feeney, Rodrigo Fonseca, Nate Foster, Brighten Godfrey, Per Gunningberg, Rob Harrison, Eric Keller, Wyatt Lloyd, Sid Sen, Jeff Terrace, Minlan Yu, the anonymous reviewers, and the paper's shepherd, Eddie Kohler, for comments on earlier versions of this paper. Funding was provided through NSF Awards #0904729 and #1040708, GENI Award #1759, the DARPA CSSG Program, an ONR Young Investigator Award, and a gift from Cisco Systems. This work does not reflect the opinions or positions of these organizations.

References

[1] BGPSEC protocol specification, draft-lepinski-bgpsec-protocol-02, 2012.

[2] IETF TRILL working group. `http://www.ietf.org/html.charters/trill-charter.html`.

[3] M. Al-Fares, S. Radhakrishnan, B. Raghavan N. Huang, and A. Vahdat. Hedera: Dynamic flow scheduling for data center networks. In *NSDI*, Apr. 2010.

[4] M. Arye. FlexMove: A protocol for flexible addressing on mobile devices. Technical Report TR-900-11, Princeton CS, June 2011.

[5] H. Balakrishnan, K. Lakshminarayanan, S. Ratnasamy, S. Shenker, I. Stoica, and M. Walfish. A layered naming architecture for the Internet. In *SIGCOMM*, Aug. 2004.

[6] D. Clark, J. Wroclawski, K. Sollins, and R. Braden. Tussle in Cyberspace: Defining tomorrow's Internet. In *SIGCOMM*, Aug. 2002.

[7] J. Day, I. Matta, and K. Mattar. Networking is IPC: A guiding principle to a better Internet. In *ReArch*, Dec. 2008.

[8] D. Farinacci, V. Fuller, D. Meyer, and D. Lewis. Locator/ID separation protocol (LISP), draft-ietf-lisp-22, Feb. 2012.

[9] A. Feldmann, L. Cittadini, W. Muhlbauer, R. Bush, and O. Maennel. HAIR: Hierarchical architecture for Internet routing. In *ReArch*, Dec. 2009.

[10] A. Ford, C. Raiciu, M. Handley, S. Barre, and J. Iyengar. Architectural Guidelines for Multipath TCP Development, Mar. 2011. RFC 6182.

[11] B. Ford and J. Iyengar. Breaking up the transport logjam. In *HotNets*, Oct. 2008.

[12] V. Fuller, D. Farinacci, D. Meyer, and D. Lewis. LISP alternative topology (LISP+ALT), draft-ietf-lisp-alt-10, Dec. 2011.

[13] A. Greenberg, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. Maltz, P. Patel, and S. Sengupta. VL2: A scalable and flexible data center network. In *SIGCOMM*, Aug. 2009.

[14] C. Kim, M. Caesar, and J. Rexford. Floodless in SEATTLE: A scalable Ethernet architecture for large enterprises. In *SIGCOMM*, Aug. 2008.

[15] T. Koponen, M. Chawla, B.-G. Chun, A. Ermolinskiy, K. H. Kim, S. Shenker, and I. Stoica. A data-oriented (and beyond) network architecture. In *SIGCOMM*, Aug. 2007.

[16] D. Mazières, M. Kaminsky, M. F. Kaashoek, and E. Witchel. Separating key management from file system security. In *SOSP*, Dec. 1999.

[17] memcached. `http://memcached.org/`, 2012.

[18] J. Mudigonda, P. Yalagandula, M. Al-Fares, and J. C. Mogul. SPAIN: COTS data-center Ethernet for multipathing over arbitrary topologies. In *NSDI*, Apr. 2010.

[19] R. N. Mysore, A. Pamboris, N. Farrington, N. Huang, P. Miri, S. Radhakrishnan, V. Subramanya, and A. Vahdat. PortLand: A scalable fault-tolerant layer 2 data center network fabric. In *SIGCOMM*, Aug. 2009.

[20] P. Natarajan, F. Baker, P. D. Amer, and J. T. Leighton. SCTP: What, why, and how. *Internet Comp.*, 13(5):81–85, 2009.

[21] P. Nikander, A. Gurtov, and T. R. Henderson. Host Identity Protocol (HIP): Connectivity, Mobility, Multi-Homing, Security, and Privacy over IPv4 and IPv6 Networks. *IEEE Comm. Surveys*, 12 (2), Apr. 2010.

[22] C. E. Perkins. IP mobility support for IPv4, RFC3344, Aug. 2002.

[23] R. Perlman. Rbridges: Transparent routing. In *INFOCOM*, Mar. 2004.

[24] B. Podmayersky. An incremental deployment strategy for Serval. Technical Report TR-903-11, Princeton CS, June 2011.

[25] U. Saif and J. M. Paluska. Service-oriented network sockets. In *MobiSys*, May 2003.

[26] A. C. Snoeren and H. Balakrishnan. An end-to-end approach to host mobility. In *MOBICOM*, Aug. 2000.

[27] I. Stoica, D. Adkins, S. Zhuang, S. Shenker, and S. Surana. Internet indirection infrastructure. *Trans. Networking*, 12(2), Apr. 2004.

[28] P. Vixie, S. Thomson, Y. Rekhter, and J. Bound. RFC 2136: Dynamic Updates in the Domain Name System, Apr. 1997.

[29] M. Walfish, H. Balakrishnan, and S. Shenker. Untangling the Web from DNS. In *NSDI*, Mar. 2004.

[30] M. Walfish, J. Stribling, M. Krohn, H. Balakrishnan, R. Morris, and S. Shenker. Middleboxes no longer considered harmful. In *OSDI*, Dec. 2004.

[31] D. Wischik, C. Raiciu, A. Greenhalgh, and M. Handley. Design, implementation and evaluation of congestion control for multipath TCP. In *NSDI*, Mar. 2011.

[32] S. Zhuang, K. Lai, I. Stoica, R. Katz, and S. Shenker. Host mobility using an Internet indirection infrastructure. In *MobiSys*, May 2003.

Reliable Client Accounting for P2P-Infrastructure Hybrids

Paarijaat Aditya[†] Mingchen Zhao[‡] Yin Lin[⋆◇]

Andreas Haeberlen[‡] Peter Druschel[†] Bruce Maggs[⋆◇] Bill Wishon[◇]

[†]Max Planck Institute for Software Systems (MPI-SWS) [‡]University of Pennsylvania [⋆]Duke University [◇]Akamai Technologies

Abstract

Content distribution networks (CDNs) have started to adopt *hybrid* designs, which employ both dedicated edge servers and resources contributed by clients. Hybrid designs combine many of the advantages of infrastructure-based and peer-to-peer systems, but they also present new challenges. This paper identifies *reliable client accounting* as one such challenge. Operators of hybrid CDNs are accountable to their customers (i.e., content providers) for the CDN's performance. Therefore, they need to offer reliable quality of service and a detailed account of content served. Service quality and accurate accounting, however, depend in part on interactions among untrusted clients. Using the Akamai NetSession client network in a case study, we demonstrate that a small number of malicious clients used in a clever attack could cause significant accounting inaccuracies.

We present a method for providing reliable accounting of client interactions in hybrid CDNs. The proposed method leverages the unique characteristics of hybrid systems to limit the loss of accounting accuracy and service quality caused by faulty or compromised clients. We also describe RCA, a system that applies this method to a commercial hybrid content-distribution network. Using trace-driven simulations, we show that RCA can detect and mitigate a variety of attacks, at the expense of a moderate increase in logging overhead.

1 Introduction

An increasing number of commercial content-distribution networks (CDNs) are based on a hybrid architecture, which combines peer-to-peer and infrastructure-based elements. These include, for instance, Velocix [5], LiveSky [35], Pando Networks [4], Octoshape [3], and PPLive [33]. Even Akamai, the largest CDN, which was originally purely infrastructure-based, has recently added peer-assisted technologies like NetSession [8]. There are good reasons for this trend. Hybrid architectures combine many of the benefits of peer-to-peer systems (cost reduction through the use of client resources, use of local resources, e.g., within a corporate intranet) with those of infrastructure-based systems (dedicated backup resources, dependability, centralized management and control); moreover, by utilizing both peer and infrastructure resources, hybrids are able to provide better service than pure peer-to-peer or infrastructure-based systems. Studies have shown impressive advantages: the potential bandwidth savings for the service provider are considerable [21], and hybrid CDNs can significantly reduce the costs of all parties involved in the content distribution process, *including* the edge ISPs [22].

However, compared to purely infrastructure-based systems, hybrid architectures face an inherent challenge: By definition, peer-to-peer communication occurs between untrusted clients, and therefore cannot be observed directly by the trusted infrastructure. As a result, faulty or compromised clients can mishandle peer communication in ways that are not observable by the infrastructure, and they can under- or overreport peer interactions. In principle, a compromised client may be able to censor or modify content, and inject unauthorized content; it may refuse, delay, or abort transfers to deny or degrade service to other clients; and it may misreport peer transfers in an attempt to manipulate the accounting for commercial content or services.

In practice, hybrid systems can take measures to mitigate this risk. For instance, clients can obtain signed content hashes from the trusted infrastructure to verify content received from peers. The infrastructure can control which clients may peer, in order to make collusion of faulty clients more difficult. Client logs can be checked and suspicious or inconsistent records excluded at some cost in logging accuracy. Clients that have repeatedly been involved in disputed or aborted transactions can be blacklisted at some risk of blocking legitimate clients. Nevertheless, the potential remains for compromised clients to disrupt service quality and affect logging accuracy.

There is little evidence of widespread attacks of this type against hybrid systems today. However, as these systems become more popular, it is important to understand the risks. Therefore, we have (with permission) performed a 'red team' evaluation of one specific hybrid CDN, Akamai's NetSession system, which has a large deployment with currently over 24 million clients. We have identified an attack vector that enables a single malicious client to report more than 30 GB of fictitious download activity per hour, an amount that can be further inflated through a Sybil attack [17].

While this specific vulnerability has been removed, the underlying challenge remains: a hybrid system's accounting is based on information from untrusted peers that is difficult to verify, and a determined attacker could find other ways to exploit this vulnerability. The challenge also applies to other commercial hybrid content delivery systems, not just to NetSession, and may apply to other types of hybrid systems as well. For example, CDNs have developed methods for owner-operated network appliances to serve customer content and report on download activity [23]. Unlike edge servers, which are part of the CDN's infrastructure, these appliances are not under the administration of the CDN. Similarly, certain network games rely on direct communication and interaction between game consoles, with outcomes ultimately reported to a centralized infrastructure [7]. Hence, we are interested in a principled approach to reliable accounting of client interactions.

To address this challenge, we present a method for providing reliable client accounting in hybrid distributed systems. Our method uses the infrastructure nodes to establish reliable facts about the clients, such as an upper bound on their available resources (which is essential to limit the effect of Sybil attacks). Moreover, all clients record a tamper-evident log [20] of their actions and must periodically upload their log to an infrastructure node; this severely restricts the ability of malicious clients to lie without getting caught.

A key feature of our approach is the ability to *quarantine* suspicious clients. Quarantined clients cannot interact with other clients, and their requests are served directly by the infrastructure; thus, such clients are unable to misreport their actions or disrupt other clients. A key insight is that *in hybrid systems, quarantining is safe*: if an honest client is accidentally quarantined, its quality of service does not change, it merely causes a small amount of extra load on the infrastructure. Thus, the infrastructure can afford to err on the side of caution e.g., by using anomaly detection techniques with high false-positive rates, which are difficult for an adversary to escape.

To demonstrate that our approach is effective, we present RCA, a system that applies our approach to NetSession. We report results from a trace-driven evaluation, based on traces from Akamai's production deployment. Our results indicate that RCA increases the protocol overhead from 0.06% without RCA to 0.47% with RCA, relative to the amount of content served, and it requires clients to maintain approximately 550 bytes of extra information per MB of downloaded content that must be uploaded to, and checked by, the infrastructure. We also show that RCA is effective against a variety of attacks and misbehaviors by malicious clients. In summary, our contributions are as follows:

- A case study of a hybrid content delivery system based on Akamai NetSession (Section 3);
- A demonstration of an accounting attack on NetSession (Section 4);
- A method for providing reliable client accounting in hybrid distributed systems (Section 5);
- RCA, an application of our approach to NetSession (Section 6); and
- A comprehensive evaluation, based on traces from NetSession (Section 7).

2 Related work

Hybrid systems: Several studies, e.g., [21, 22], have predicted considerable benefits for peer-assisted CDN designs, and measurement studies of commercial peer-assisted CDNs, such as LiveSky [35] and PPlive [33], seem to confirm that these benefits are being achieved in practice. Client misbehavior in peer-to-peer CDNs has been observed empirically, e.g., collusion among users of the Maze system [24]. Most existing defenses assume rational clients, who misbehave to increase their own performance or minimize their cost. Misbehavior of this type can be prevented with robust incentives; for instance, Dandelion [31] rewards uploads from clients with virtual currency that can be used to purchase downloads from other clients or the infrastructure. However, such incentives are not effective against malicious attacks of the type we consider in this paper. Dandelion focuses on preventing freeloading, while RCA considers more general Byzantine behavior as well.

Some systems do consider certain types of malicious behavior. For example, Antfarm [28], which uses cryptographically signed tokens as payment for block downloads, can detect forged tokens; [28] also briefly sketches possible extensions that could mitigate additional attacks, such as double-spending. However, Antfarm does not use the tokens for reporting, but rather for allocating the infrastructure's bandwidth among the different swarms, and consequently has lower requirements for the accuracy of reporting. Also, Antfarm's weapon against malicious clients is to excise them from the swarm. Unlike quarantined clients in RCA, excised clients no longer receive service. Therefore, clients can be excised only in cases of cryptographically verifiable misbehavior, such as forged tokens or double-spending.

Accounting mechanisms: Seuken and Parkes [30] examined the problem of reliable accounting in a decentralized setting with rational peers, and they have shown that, in this setting, no Sybil-proof accounting mechanism exists. This result does not apply to hybrid systems because the infrastructure can serve as a central, trusted monitoring component. Moreover, [30] assumes that peers are either cooperative or rational, whereas this paper also considers Byzantine peers.

Accountability: Accountability systems like PeerReview [20] can automatically detect a large subclass of Byzantine faults and tie them to the identities of specific faulty nodes. RCA's tamper-evident log is based on ideas from PeerReview, but differs from PeerReview in several ways; for example, RCA is designed to withstand Sybil attacks, and takes advantage of a central infrastructure in order to reduce overhead. Also, RCA's focus is on reliable accounting rather than fault detection.

Defenses against Sybil attacks: Douceur's original paper on the Sybil attack [17] suggests resource testing as a possible defense, and subsequent work has explored a variety of solutions for different types of resources, such as distinct physical locations [10], money [25], and social relationships with honest users [36, 37]. Our approach is perhaps closest to Tarzan [18], which uses IP addresses to identify instances of Sybil identities. Another approach relies on certification authorities, as in [6]. Certification is unsuitable for systems like NetSession, which seek to keep the registration process simple to encourage adoption.

Anomaly detection: RCA includes anomaly detection [13] and benefits in this regard from decades of research, e.g., on intrusion-detection systems [15]. However, when applied by itself, anomaly detection faces limitations in a security context [9], because an adversary may be able to avoid detection by shifting the system's workload gradually, e.g., in a frog-boiling attack [12]. To guard against such attacks, RCA complements anomaly detection with tamper-evident logs as well as consistency and invariant checks, which do not rely on assumptions about the system's workload.

3 Case study: Akamai NetSession

To provide some context for our discussion of reliable accounting in hybrid systems, we first describe the design of a concrete hybrid system: Akamai NetSession.

3.1 The NetSession system

The NetSession system is a peer-assisted content delivery network (CDN) operated by Akamai. Like Akamai's more widely known infrastructure-based service, NetSession's primary function is to accept *content*, such as software packages or videos, from a number of *content providers*, and to deliver that content to a potentially large number of *users*. While the infrastructure-based service delivers the content exclusively from a number of *edge servers* that are operated by Akamai, NetSession additionally leverages peer-to-peer transfers between the user nodes, or *clients*. Content can be downloaded from the edge servers, the peers, or a mixture of both.

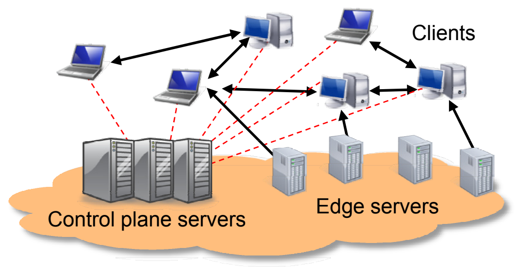


Figure 1: Overview of the NetSession system. Control connections are shown as dotted lines.

To use NetSession, users must install the NetSession client software on their machines. The NetSession software can maintain links to other clients and communicates with a number of *control plane* servers, which coordinate the connections between the clients (Figure 1). Once installed, the client can be reused for future downloads, and it offers an API to local applications. The client also provides a GUI for monitoring download progress, as well as a set of controls for enabling or disabling uploads to peers. Clients are identified by a GUID, which is chosen at random during installation and does not contain personally identifiable information. Since each client periodically connects to the control plane, we can use the number of such connections to estimate the number of clients in the system. During November 2011, the control plane registered connections from more than 24 million distinct GUIDs.

3.2 Why a hybrid architecture?

Historically, most CDNs have *either* been infrastructure-based, such as Akamai and Limelight [2], *or* peer-to-peer, such as PPLive [33], Joost [1], or BitTorrent [14]. Both architectures have their own advantages and disadvantages. The key advantage of peer-to-peer systems is their scalability and independence of infrastructure; infrastructure-based CDNs can only scale by provisioning additional infrastructure resources, which can be expensive. The key advantage of infrastructure-based systems is their predictable quality of service; the service provided by peer-to-peer systems, on the other hand, can be inconsistent [34], because end-user machines are often resource-constrained and unreliable.

Hybrid systems seem like an attractive design point because they can combine most of the advantages of both systems. On the one hand, they can scale and reduce cost by leveraging resources contributed by the clients; on the other hand, they can mask glitches and performance problems by falling back on the infrastructure, which ensures good quality of service. In addition, the infrastructure can be used to solve a variety of technical problems that tend to plague peer-to-peer systems. For example, it can serve as a rendezvous point for NAT traversal, it can perform global optimizations [28], it can provide extra capacity in the critical initial and final stages of the download, and it can validate content exchanged among clients. The infrastructure can also maintain a central directory, so that, when a client initiates a download, the infrastructure can immediately suggest a number of peers that are online, have a compatible NAT type, and are storing the requested file.

However, to secure hybrid systems, we need to understand whether their specific structure makes them vulnerable to new kinds of attacks, and what kinds of defenses may be appropriate for them. In this paper, we identify a new class of attacks that exploit the infrastructure's inability to observe client interactions directly, and we present a system that can detect and mitigate these (and other) attacks.

3.3 Operation

We now briefly describe how a peer-assisted download in NetSession works. When the user of a client c initiates a download, the NetSession software on c sends a request to the control plane, which returns a set of up to k peers (currently 40) $C := c_1, \ldots, c_k$ that are currently online and have blocks of the requested file. The list contains only peers that have compatible NATs and is biased towards peers that are close to c. Once c has received this list, it opens connections to some subset of C. The control plane also notifies the peers in C, so that both endpoints can open holes in their local NATs if necessary.

The actual download is done using a swarming protocol conceptually similar to the BitTorrent [14] protocol: the file is broken into fixed-size blocks, clients exchange bitmasks with their peers to indicate which blocks they have available, and clients can request ranges of blocks from each other. Swarming and downloads from edge servers can proceed concurrently. Each block is verified against a hash that is obtained from an edge server; thus, corrupted blocks can be discarded and downloaded again.

Clients only upload content file blocks to their peers when three conditions are met: 1) the local client has previously downloaded the file, 2) the content provider has enabled swarming for that file, and 3) swarming is enabled in the client's local NetSession installation (this can be changed by the user in a control panel setting). To avoid inconveniencing the user, NetSession limits the upload/download ratio, and it ensures that uploads do not cause resource contention on the local machine.

NetSession downloads are peer-*assisted* in the sense that downloading content from peers is helpful but not required for correctness: even if a client does not receive a single valid block from its peers (or there are currently no peers that have the requested content), it can still complete the download using only the edge servers. Nevertheless, it is important to identify faulty and misbehaving clients, since they can degrade the CDN's quality of service. For instance, faulty clients can request an inordinate number of blocks from their peers or send them corrupted data blocks, which subsequently fail the hash verification and must be downloaded again. This is not an issue in infrastructure-based CDNs, where all content is delivered by the edge servers.

3.4 Logging and reporting

NetSession generates a detailed log to document its performance. Each client regularly reports certain events to the control plane, including 1) the start and the completion of a download, 2) its performance during the download, and 3) the number of bytes that it has downloaded from peers and from the infrastructure. These reports are logged by the control plane.

The NetSession logs are used internally by Akamai, e.g., for quality control and to improve system performance. However, they also serve another crucial function. Customers of CDNs expect to have access to the logs of all downloads of their content; they use these logs for analytics, i.e., to determine which content is popular, which clients are downloading the content, and what level of performance the CDN is providing to the clients. Since the logs are directly visible to the customer, it is essential (from the CDN operator's perspective) that the logged data is reliable. If the data in the logs were found to be inaccurate or subject to manipulation by users, this could undermine the CDN operator's credibility.

We hypothesize that logging and reporting are key features of hybrid systems in general, beyond the specific NetSession system, and perhaps even beyond hybrid CDNs. By definition, a hybrid system has an infrastructure component, which must be paid for by some party, and that party will want an accurate report of what they are paying for. Hence, attacks on reporting represent a serious threat to hybrid systems.

4 Attacks on hybrid systems

In this section, we describe attacks on hybrid CDNs, including a novel class of *inflation attacks* on the reporting system. To demonstrate that existing systems are vulnerable, we report results from a successful inflation attack on the NetSession system.

4.1 Threat model

In this paper, we assume that the nodes in the infrastructure (such as Akamai's edge servers) are correct and fully trusted by the operator. The clients, on the other hand, are untrusted and can be compromised or fail in various ways, so we conservatively assume that some subset of them is controlled by a malicious adversary.

Clients can communicate with infrastructure nodes and with their peers, but the infrastructure cannot observe direct peer-to-peer communication.

We also assume that the system does not use strong identities and that membership is open, i.e., any client is allowed to join the system. In particular, this means that the adversary can potentially mount a Sybil attack [17] on the system, e.g., by running multiple instances of the client software on the same machine.

4.2 Attack vectors

There are two aspects of this model that make reliable accounting fundamentally difficult. First, the clients are not fully trusted and could tell lies, suppress information, or mishandle the communication with their peers to deny them service. Second, a (potentially large) fraction of the events that are to be accounted for occur directly between the clients, where the infrastructure cannot observe them. Separately, each of these challenges would be easy to handle: if the clients were trusted, the infrastructure could rely on their correct handling of requests and the accuracy of their reports; if the infrastructure was directly involved in all communication (as in Akamai's infrastructure-based CDN), it could intercede when clients misbehave towards their peers, and perform accounting based on its own records.

In the following, we focus on a novel type of accounting attack that exploits these challenges. We will refer to this attack as an *inflation attack*. In an inflation attack, the attacker causes the system to overreport the amount of service that it has provided. Using the same technical approach, one could also mount *deflation attacks*, in which the attacker would cause the amount of service to be underreported instead.

4.3 Inflation attack on NetSession

To demonstrate the potential impact of inflation attacks, we carried out such an attack on the NetSession system. The attack could be carried out easily by modifying the NetSession client software. We decided against this approach, partly because a modified client might have accidentally disrupted other clients or the NetSession infrastructure, partly because we were able to carry out the attack even with an unmodified NetSession client.

To accomplish this, we wrote a script that uses the client's API to repeatedly download a certain file, as well as a small proxy that interposed between our local NetSession client and its peers. Whenever our client attempts to contact a peer, our proxy returns a spoofed response that indicates that the peer has all of the requested blocks, and whenever our client requests any blocks, the proxy returns the blocks at LAN speed. This man-in-the-middle attack was possible because, unlike the messages exchanged between clients and the infrastructure, communication between peers was not signed. The effect of this attack was that our client would trigger a large number of downloads, and these downloads would complete at LAN speed. The resulting large number of downloads were reported to the infrastructure, even though no content was actually transferred among the peers.

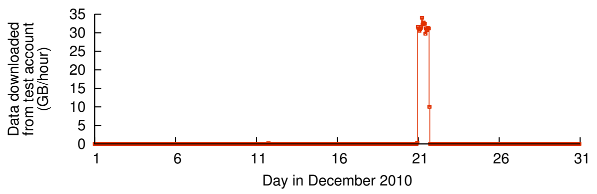


Figure 2: Effect of our attack on the NetSession logs.

4.4 Impact of the attack

We obtained permission from Akamai to test our attack on the deployed NetSession system. To avoid the risk of interfering with the production system, we targeted a set of special files that is normally used for testing; thus, there was no risk that legitimate clients would attempt to download files from our proxied client, or that our attack would affect the logs of legitimate content providers. We ran our attack for one full day; during that day, we requested files as quickly as possible. To assess the impact of the attack, Akamai gave us access to the control plane logs for our client's specific GUID.

Figure 2 shows the reported downloads in this log. Our single modified client was able to generate about 30 GB/hour of fictitious downloads, which show up as a sharp spike in the figure. We note that this is a proof-of-concept attack, and that its throughput was limited by the throughput of our proxy. Had we chosen to directly modify the client software, we could have reported downloads at arbitrary rates.

4.5 How serious is this attack?

Our specific proof-of-concept attack is not difficult to prevent; indeed, the attack as described no longer works. However, our specific attack is just one example from an entire class of attacks; for instance, we could have reverse-engineered the NetSession client software and directly modified the reports it sends to the NetSession infrastructure. So the root of the problem runs deeper.

Moreover, the vulnerability seems to exist in hybrid systems generally; it is inherent in the fact that hybrid systems must account for interactions between untrusted clients, who cannot be relied upon to report them accurately. To prevent similar attacks on NetSession and other hybrid systems, a more comprehensive solution is needed. Also, to show that a simple fix is not sufficient, we briefly discuss a few strawman solutions.

Sign all messages: Cryptography can be used to defend against man-in-the-middle attacks, but recall that this was just a trick we used to make our proof-of-concept attack easier to implement. Instead, an attacker could simply modify the client software to generate whatever messages he needs it to send.

Detect software modifications: There are techniques that aim to detect whether a client has modified software. Some of these techniques are purely software-based; for example, certain multiplayer games scan the clients' memory for known cheats [29]. However, an adversary can circumvent these techniques by disabling the scan or by reporting incorrect results. Other techniques require hardware support; for example, trusted platform modules can be used to certify that a client has loaded a certain software image [19]. However, the requisite hardware usually is not available on all clients, and even where it is available, it may be able to certify only that a certain binary was loaded, not that it is still running.

Limit clients to one download per file: This would thwart our specific attack; however, an adversary could still download many different files or create many Sybil identities that each download the file only once.

Anomaly detection: A massive load spike like the one in Figure 2 would probably raise the CDN operator's suspicion. However, there could be a legitimate reason for the spike, and there is no way to establish its provenance after the fact. Thus, the operator is caught in a dilemma: if the spike is genuine, it is probably important for the content provider to know about it, so it must be left in the log; if the spike is fake, its presence distorts the accounting, so it should be removed.

5 Reliable accounting

In this section, we describe our method for providing reliable accounting in hybrid systems. Although we developed this method with NetSession in mind, it should be applicable to other types of hybrid systems (such as P2P streaming or storage systems). We begin with a description of the general approach and then show (in Section 6) how it can be applied to NetSession.

5.1 System model

We consider a system that consists of a number of trusted infrastructure nodes and a (potentially much larger) number of untrusted clients. The system offers a service to the clients that can be provided either by the infrastructure nodes or by other clients. The infrastructure cannot directly observe interactions between the clients.

Our goal is to provide an *accounting mechanism* that reliably captures client activity. Specifically, we are interested in activity by faulty or compromised clients that could degrade the performance of the system or distort the record of services actually rendered.

F1	Fail to log exact set of messages sent or acknowledged	5.4
F2	Fail to log consistent sequence of messages	5.5
F3	Execute illegal, or fail to execute required, protocol action	5.6
F4	Faulty peers collude to report fictitious exchanges	5.7, 5.8
F5	Render poor service to peers	5.8
F6	Nefarious user requests	5.9
F7	Sybil attack	5.10

Table 1: Types of client misbehaviors. The last column shows the subsection that describes the countermeasure.

5.2 Threat analysis

Next, we characterize the kinds of threats that a faulty or malicious client poses to the rest of the system. To do this in a protocol-independent way, we model the client as an abstract state machine that accepts requests from the local user (e.g., names of files to download) and eventually produces responses (e.g., the contents of the file). The state machine can send and receive messages, and it must periodically upload a log of its actions to the infrastructure. Each state machine is expected to follow a specific protocol, and this protocol is not necessarily fully deterministic (e.g., clients might be allowed to choose the peers with the highest throughput).

Table 1 summarizes the types of threats we consider here. Faulty or malicious clients can *fail to log* the exact set of messages they have sent or acknowledged (F1), or fail to log them in a sequence that is causally consistent (F2). They can *violate the protocol*, either by making a bad state transition or by failing to make a required state transition (F3). They can collude with faulty peers in order to report fictitious transactions amongst each other (F4). They can *deliver poor performance*, e.g., by being slow to respond to messages from peers, by aborting or delaying peer transfers, or by sending corrupted content (F5). A user can issue *nefarious content requests* in order to create artificial demand for a provider's content or to degrade the service quality enjoyed by other clients (F6). Finally, a user can mount a *Sybil attack* by joining the system under more than one identity, in order to amplify other attacks (F7).

5.3 Approach

In a fully decentralized system, it is difficult or even impossible [30] to provide reliable accounting. In a hybrid system, however, we can do better by leveraging some of the unique characteristics of these systems, namely:

1. **Trusted infrastructure:** The nodes in the infrastructure are directly controlled by the operator;
2. **Central control:** The operator can prescribe a single protocol that all the nodes must follow;

3. **Global view:** The operator is able to observe the status of all clients eventually; and
4. **Dedicated resources:** The infrastructure has the capacity to take over for under-performing or suspicious clients.

In the following, we describe a sequence of techniques for constructing a reliable accounting mechanism that takes advantage of these characteristics. Each technique adds some constraints on the kinds of behaviors a faulty node can manifest without getting caught.

5.4 Require message commitment

First, we require clients to cryptographically sign all messages and acknowledgments they send. Moreover, clients record the signatures of received messages and acknowledgments in a log, which they periodically forward to the infrastructure. As a result, a faulty client can no longer deny having sent or received a message it has previously sent or acknowledged. Likewise, a faulty client cannot falsely claim that a correct client has sent or acknowledged a message, because the faulty client would not be able to produce the corresponding signature.

5.5 Check logs for consistency

Even with message commitment in place, a faulty client could give a false account of the order in which certain events happened. For instance, a client could receive an object from the infrastructure, acknowledge it, and then later decline a peer's request for the same object. In its logged record, it could claim that the peer's request had arrived before it received the object from the infrastructure, in order to justify its failure to serve the object.

This form of misrepresentation can be avoided by requiring each client to maintain a *tamper-evident log* [20] of its actions. For this purpose, the log entries form a hash chain, and the hash of the most recent log entry is included in the signature of any message that the node signs. Thus, the node commits to its entire event history each time it sends a message or acknowledgment. Because the logs and all of a node's signatures are eventually sent to and checked by the infrastructure, a client would be caught if it ever omitted, fabricated, or manipulated events in its logs, or gave inconsistent accounts of the sequence of events.

Each client is forced to log a single linear account of its actions that includes all acknowledged messages sent to, or received from, correct peers. If messages are forged, omitted, reordered, or tampered with, the client effectively makes a signed admission of guilt.

5.6 Check logs for plausibility

Even when a client's log is consistent with the logs of all other clients it has communicated with, the log can still be implausible; for example, a client A might download a file from a peer B and then, when a third peer C requests that file from A, serve a modified version of the file or claim that it no longer stores the file. To prevent this, our second step is to verify that a log is *plausible*, i.e., it is consistent with a valid execution of the software the client is expected to run.

We can decide whether a log is plausible by checking that it satisfies a set of *invariants*, which must hold in any correct execution of the client software. Typical example invariants state that a client must only serve content it has previously received, may only contact or accept requests from peers suggested by the infrastructure, etc.

5.7 Control client pairings

If clients are free to choose which clients to request services from, malicious clients can collude to request services from each other (and thus make consistent and plausible logs without actually doing any work), or they can 'gang up' on some correct clients to deny them services. To make collusion more difficult, the infrastructure can impose *restrictions* on the clients, e.g., by limiting each client i to only request services from peers in some set S_i.

Restrictions should be neither too tight nor too loose. In the above example, very small sets S_i will force the clients to contact the infrastructure frequently (e.g., to ask for additional clients if the ones in S_i fail), which increases overhead and decreases performance, whereas very large sets S_i increase the chances that a malicious client i will find an accomplice in its set S_i.

The infrastructure's choice of S_i may depend on which peers have the requested content, which ISP a peer is connected to, and whether it has a compatible type of NAT. Some of these factors can be influenced by a peer; for instance, a peer could download rarely requested content in order to increase the chance that it will be paired with a colluding peer that requests that same content. Therefore, controlling client pairing can only mitigate but not eliminate client collusion.

5.8 Quarantine anomalous clients

A faulty client can degrade the service received by its peers, e.g., by providing the requested services inconsistently or too slowly. Moreover, as discussed above, colluding peers could inflate their upload activity. Our next step is to apply statistical *anomaly detection* to identify potentially problematic clients, and to quarantine those clients.

Ordinarily, anomaly detection systems face a difficult tradeoff between effectivity and the number of false positives. Our key insight is that, in the specific case of hybrid systems, *this tradeoff can be avoided by redirecting any suspicious clients to the infrastructure*. In other words, when a client c manifests anomalous behavior, the infrastructure can restrict c to contacting only trusted infrastructure nodes, and it can tell other clients not to contact c.

If c is malicious, quarantining c will ensure accurate logging because a) c's interactions with the infrastructure will be logged by the trusted nodes, and b) c cannot plausibly log any interactions with other nodes. On the other hand, if c is correct and its detection was a false positive, c's requests will still be handled by the infrastructure, so c's user will still receive good service. Since quarantining clients is 'safe' from a QoS perspective, we can perform anomaly detection aggressively and accept a nontrivial false-positive rate, as long as the infrastructure has sufficient resources to handle the extra load.

5.9 Flag/throttle suspicious user behavior

In addition to the types of client misbehaviors covered in the previous section, there is a class of attacks that is caused solely by user activity, while the client software behaves as expected. For instance, a user could nefariously download content from a specific content provider, in order to drive up demand for that provider's content. (This type of attack would typically be combined with a Sybil attack and possibly a botnet. Also, note that this attack is not specific to hybrid CDNs.)

Based on the tamper-evident logs collected by the system, we can perform statistical anomaly detection to identify clients whose download activity stands out in terms of volume and content selection. A flagged client can be immediately subjected to a download rate-limit by the infrastructure, pending resolution by a human operator. At the same time, a human operator is notified of the anomaly for further inspection and resolution. For instance, an operator can contact a content provider to check if a sudden increase in demand (possibly from a specific set of IP addresses) is expected.

5.10 Enforce resource limits

Some of the attacks described in the previous sections can be amplified by a *Sybil attack*, where an attacker registers more than one instance of the client software for each physical node he controls. For instance, the impact of a colluding-peers attack or a nefarious download attack increases with the number of client instances. Without strong user identification, we cannot effectively prevent Sybil attacks, but we can at least constrain the aggregate amount of service that Sybils can log. For example, we can check that, in aggregate, the activities recorded in these logs cannot exceed the physical capacity of the adversary's nodes.

Since a hybrid system contains trusted infrastructure nodes, we can achieve this goal through resource testing. For example, a client could be required to demonstrate its upstream or downstream bandwidth in a short data exchange with the infrastructure. The infrastructure could refuse to accept multiple clients with the same IP address, and/or ask a group of clients with a common IP prefix to demonstrate that they run on separate machines by asking each of them to simultaneously solve a different crypto puzzle. The results could then be used to flag implausible client activity, such as clients who claim to have exchanged data with peers in different networks at a rate that exceeds their measured access link capacity, or clients who claim to have exchanged data with a number of clients on the same network that exceeds the number of separate machines they have demonstrated.

6 Application to NetSession

In this section, we describe the RCA system, which applies the method from Section 5 to NetSession.

6.1 Overview

Our design instantiates each of the building blocks we have presented in Section 5. We use resource certificates (Section 6.3) to limit the aggregate bandwidth of Sybils, a novel implementation of tamper-evident logs that has been optimized for hybrid systems (Sections 6.4 and 6.5) to check for consistency, a set of NetSession-specific invariants (Section 6.6) to check for plausibility, and a set of statistical tests (Section 6.7) for quarantining anomalous clients. The rest of this section describes each of these building blocks in more detail.

The basic workflow in RCA is as follows. When a client i first joins RCA, it contacts one of the control plane servers and uploads a short file to demonstrate its link capacity; the control plane then issues the client a private key σ_i (for signing messages), a public key π_i, and a certificate Γ_i that encodes the measured capacity. The client can then download or upload content, just as in the original NetSession system, but it additionally maintains a tamper-evident log, which it periodically uploads to the control plane. The control plane forwards the logs to a set of backend servers, which process them and produce the accounting information. The control plane also applies statistical tests to detect and quarantine anomalous clients.

6.2 Assumptions

The design of RCA relies on the following assumptions:

1. Infrastructure nodes are trusted by the operator and can only fail by crashing.
2. All nodes have access to a cryptographic hash function H.
3. Faulty clients cannot forge the signature of correct peers or of the infrastructure.

Assumption 1 seems reasonable in a centrally managed CDN like NetSession; assumptions 2 and 3 are commonly assumed to hold for hash functions like SHA-256 and algorithms such as RSA, provided that the keys are sufficiently strong.

6.3 Resource certificates

To prevent malicious clients from reporting more activity than they physically have the capacity to perform, RCA uses a few simple resource tests, as discussed in Section 5.10.

When a client i first joins the system, it contacts one of the control plane servers and requests a key pair, which will constitute the client's identity for the purposes of reporting. The control plane then exchanges some amount of data with the client, and it measures the maximum throughput C_i that i achieves during the exchange.[1] Finally, the control plane then generates a fresh key pair σ_i/π_i and returns it to the client, along with a certificate $\sigma_P(\pi_i, G_i, C_i, A_i, T_i)$ that is signed with the control plane server's private key σ_P and binds the client's public key π_i to the client's GUID G_i, its measured capacity C_i, its IP address A_i, and an expiration time T_i (on the order of a few hours). When a client's current certificate expires or its IP address changes, the client repeats this process to obtain a fresh certificate.

To prevent Sybil attackers from obtaining multiple certificates for the same IP address, the control plane internally maintains a table with all unexpired certificates. Suppose a client c requests a new certificate from an address A_k while there are still unexpired certificates for A_k. Then the control plane revokes[2] any certificates for A_k whose clients are not currently logged in, and it asks the remaining clients to upload *at the same time* as c. It then measures the aggregate bandwidth C of all the uploads and issues c a certificate for the difference between C and the sum of the capacities from the existing certificates. To defend this mechanism against malicious clients that attempt to overload the control plane with requests for new certificates, clients can be required to solve a puzzle [27] before submitting a request; the difficulty of the puzzle can be a function of the current load on the control plane.

In summary, the infrastructure ensures that there can be only one valid certificate per IP address at a time, and that an adversary with an aggregate capacity C cannot obtain certificates whose aggregate capacity exceeds C. Additional resource tests could be implemented and the results included in the resource certificate.

[1]Note that this requires two-way communication and thus prevents malicious clients from obtaining certificates for spoofed addresses.

[2]In our setting, revocation is comparatively easy because each client has to show its current certificate to the control plane when logging in.

6.4 Tamper-evident log

RCA requires clients to maintain a tamper-evident log of all the messages they send and receive. Unlike previous implementations designed for decentralized systems [20], a hybrid system requires different tradeoffs. On the one hand, logs are audited exclusively by the infrastructure, which simplifies the implementation. On the other hand, we need to aggressively minimize the overhead for the infrastructure—particularly the number of cryptographic signatures it has to verify—since we expect the number of clients to be orders of magnitude higher than the number of infrastructure nodes.

Each client maintains a log of entries $e_k := (h_k, s_k, t_k, c_k)$, where h_k is a hash value, s_k a sequence number, t_k an entry type (SEND or RECV), and c_k some type-specific content. The hash values form a hash chain of the form $h_k := H(h_{k-1}||s_k||t_k||c_k)$. Whenever a node i sends a message, it must attach an *authenticator* $(s_k, h_k, \sigma_i(s_k\,||\,h_k))$, which is signed with i's private key σ_i and represents a commitment to the current state of i's hash chain. Each message must be acknowledged, and clients may have at most $\mathrm{n}_{\mathrm{max}}$ unacknowledged messages in flight at any given point in time. Finally, each message or acknowledgment contains enough information to verify that its transmission has been recorded in the log. If i forges, omits, or tampers with log entries after the fact, the infrastructure can detect this by comparing the log i has uploaded to the authenticators i has sent to its peers.

RCA's tamper-evident log maintains sub-chains for each pair of communicating peers. As a result, RCA's authenticators are cumulative, i.e., the authenticator in a message or acknowledgment from i can be used to verify all previous messages or acknowledgments from i, respectively. Hence, each client need only keep one pair of authenticators for each other peer it has communicated with–rather than one for each message it has sent or received–which dramatically reduces the number of authenticators that must be uploaded to, and verified by, the infrastructure.

In summary, the tamper-evident log ensures that inconsistencies between logs can be attributed to a specific misbehaving client.

6.5 Consistency checking

When a client i is ready to upload its log λ_i, it signs λ_i with its private key σ_i, and it attaches its certificate Γ_i as well as the set A_i of authenticators it has collected from other nodes. The infrastructure must then check the log for consistency and plausibility.

At first glance, the consistency check seems to require cross-checking logs from different clients. However, RCA's tamper-evident log is structured such that, for each message transmission, *both* endpoints have sufficient evidence (specifically, an authenticator from the message or its acknowledgment) to show that the entry in the local endpoint's log is consistent with the entry in the remote endpoint's log. Thus, logs can be checked individually, which makes the log checking both efficient and trivially scalable.

Although i uploads the above information to a specific infrastructure node, other infrastructure nodes may also require information about i, namely authenticators or a certificate revocation. Before checking can begin, all information about i must be collected at a single node $H(i)$, which can be chosen, e.g., via consistent hashing. This requires a 'shuffle' step (similar to the one in MapReduce) in which each infrastructure node sends copies of its received authenticators to the nodes that are 'responsible' for them.

Next, the infrastructure inspects λ_i and checks whether a) the log is well-formed and signed with σ_i; b) the certificate Γ_i is valid and matches σ_i; c) Γ_i was not expired or revoked when the log was signed, d) at no point in the log were there more than $n_{\max}$ unacknowledged messages, e) each of the sub-hashchains is intact, f) the end of each sub-hashchain corresponds to one of the authenticators uploaded by i, and g) λ_i is consistent with all authenticators signed with σ_i. The last check can be done incrementally if more authenticators are uploaded. If any of the above checks fails, i is clearly faulty. The GUID of a faulty client is immediately disabled so that it cannot participate in peer-to-peer transactions or infrastructure downloads; also, the system operator is notified.

6.6 Plausibility checking

When a client's log passes the consistency check, we know that its recorded sequence of messages is consistent with the log of other clients. However, the messages in the log do not necessarily correspond to a valid execution of the client software. To detect misbehaving nodes, RCA checks each log to see if it satisfies the following invariants, which capture the essence of RCA's swarming protocol. Specifically, clients

1. may only exchange data with peers or edge servers that were suggested to them by the infrastructure;
2. may only serve data they have already downloaded;
3. must not modify blocks before serving them;
4. must serve blocks they have available (i.e., blocks they store and for which peering is enabled); and
5. may only request blocks they do not already store.

If the log does not satisfy all of the invariants, RCA disables the GUID of the client and notifies the system operator. Otherwise, RCA identifies, for each uploaded data block, the content provider that owns the corresponding file, and it tallies, for each content provider, the number of bytes that were uploaded on its behalf, minus any bytes that would exceed the bandwidth in the client's resource certificate.

6.7 Statistical tests and quarantine

RCA's control plane continually maintains statistics about the download and upload activities of each client, such as its IP address, its geolocation, or the number of bytes downloaded and uploaded during the last k days. The control plane uses this data and a set of statistical tests to identify anomalous clients.

When a client i is flagged as anomalous, the infrastructure quarantines i and redirects any future download requests from i to the infrastructure nodes. The client will continue to receive service, however; RCA merely ensures that any interactions with i involve at least one trusted endpoint, so i's actions can be accounted accurately. (Logs produced by i before the quarantine will still be accepted, provided that they pass all the other tests.) In other cases, the infrastructure merely notifies a human operator for resolution.

There are many kinds of statistical tests that could be useful. In Section 7.7, we describe and validate a small set of statistical tests for the NetSession system, and in Section 8, we discuss other tests that could be applied.

6.8 Limitations

RCA's tamper-evident log is only guaranteed to detect inconsistencies in message exchanges when at least one of the two endpoints is an honest node; if the infrastructure (unknowingly) pairs up two colluding clients, one of them can claim to have downloaded a large part of the file from the other without actually having done so. Controlling client pairing, applying statistical tests, and quarantining help to mitigate this limitation.

A second limitation is related to the use of anomaly detection. Since the operator usually does not know which clients are malicious, he can only base the statistical test on the observed behavior of all clients, which is only safe as long as the fraction f of clients controlled by a single adversary is small. If f is large, the adversary can slowly change the behavior of his clients over the course of several weeks or months, analogous to a frog-boiling attack [12]; this might prompt the operator to adjust the statistical tests, which would progressively relax the constraints on the adversary. However, we expect that in practice, few adversaries would have both the required number of clients and the necessary patience.

7 Evaluation

To evaluate RCA, we implemented a clone of the NetSession client and infrastructure software, called

NetSession-Base, which includes all functionality required for our experiments. NetSession-Base is complete enough to run on the Internet. However, we perform most of our experiments in a network emulation environment, which can run hundreds of NetSession-Base clients on a single machine. The network emulator models bandwidth (upstream and downstream) and propagation delay, but not packet loss. Emulations are driven by a trace that defines the node characteristics (geolocation, link capacities, IP and GUIDs) as well as the workload, i.e., the downloads and their precise timing.

We then added RCA's defenses, including the tamper-evident log, the consistency and plausibility checks, the statistical checks, and the client quarantine. We use RSA with 1024-bit keys for the cryptographic signatures. We will refer to the system with defenses enabled as NetSession-RCA.

7.1 Validation

The goal of our first experiment is to verify that our clone matches the behavior of the Akamai NetSession system closely enough so that we can use the NetSession-Base system as a baseline in subsequent experiments. For this purpose, we used Akamai's NetSession client to download a 760 MB file in the live system, and used Wireshark to capture the network traffic from and to the client.

From the captured network traffic, we then compiled a trace that replicates this download in the emulator, such that the same proportion of data are downloaded from the same number of peers and the infrastructure. We then ran NetSession-Base using this trace, and we measured the client's control and data traffic exchanged with each peer and the infrastructure. The results were all within 1% of those obtained with Akamai's NetSession.

7.2 Experimental setup

For the following experiments, we used a 30-day trace from Akamai's live NetSession system recorded in December, 2010. The trace includes an identifier and size for each object requested, the time when the download was initiated and completed, and the number of bytes downloaded from other peers. Our network emulator cannot scale to the entire workload recorded in the trace, so we used a sample that includes all downloads initiated by a randomly chosen subset of 500 clients.

We assigned client link capacities by randomly sampling from a measured distribution of download and upload speeds in residential broadband networks [16].

The NetSession traces do not record how many and which peers were actually used in a download, or how many bytes were obtained from each. In any case, because our emulation only includes a small sample of the actual peers, it would not include many of the peers who

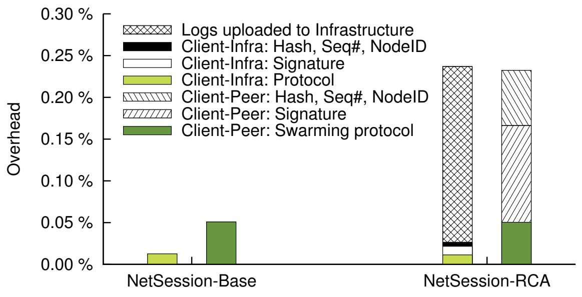


Figure 3: Network traffic overhead, normalized to the size of the downloaded content. For each system, the figure shows the communication with the infrastructure (left bar) and with peers (right bar).

actually uploaded to one of the peers in the sample. Instead, our emulation assumes that all peers in our sample can serve all files to other peers. The infrastructure suggests a random set of emulated peers for each download, and the swarming protocol dynamically requests content from this set of peers based on the observed bandwidth. Since our evaluation is not concerned with the dynamics of the swarming protocol, this approximation does not affect the results. To be conservative, we fixed the overall ratio of bytes downloaded from peers versus bytes downloaded from the infrastructure to 80%; based on our observations from the trace, this will overestimate the overhead of our system.

7.3 Cost: Traffic

To quantify the additional bandwidth requirements of NetSession-RCA, we measured a) the total number of bytes downloaded as actual payload by all clients, and b) total number of bytes transmitted in the system. The difference between the two numbers is an estimate of the bandwidth overhead of the system relative to the actual payload bytes; it was 0.06% for NetSession-Base and 0.47% for NetSession-RCA. This amounts to a 7.8-fold increase in bandwidth overhead for NetSession-RCA.

While the relative increase in overhead is substantial, it is important to note that the absolute bandwidth requirement is still modest. The average per-peer bandwidth requirement is only 192 KB/day for NetSession-RCA and 26 KB/day for NetSession-Base. In return, NetSession-RCA provides much more fine-grained and reliable information about peer behavior than NetSession-Base.

Figure 3 shows a more detailed breakdown of the results. Both NetSession-RCA and NetSession-Base exchange some control messages with peers and with the infrastructure; the corresponding amount of traffic is small and identical in both systems. RCA adds an authenticator and an acknowledgment for each message. The overhead is higher for the infrastructure traffic because the number of messages is higher: in addition to the data blocks, this traffic also contains a number of small control messages. Finally, clients must upload their logs to the infrastructure.

7.4 Cost: CPU

NetSession-RCA requires more client CPU than NetSession-Base, because it must generate and verify the signatures in authenticators. Because NetSession is intended to run in the background without inconveniencing the user, this additional computation must not consume more than a small fraction of the CPU.

To estimate the cost, we measured the number of signature generations and verifications performed by clients as part of the download activity. We then benchmarked RSA-1024 signature generation and verification on a single core of an Intel Xeon X5650 CPU, and we used these benchmarks to estimate the additional CPU load that would be caused by these operations. The maximum additional CPU load over all clients was never more than 0.5%.

7.5 Cost: Log storage and log upload

NetSession requires each client to maintain a log, and to periodically upload this log to the control plane. However, NetSession-RCA's log is considerably more detailed because it keeps track of individual messages, whereas NetSession-Base's log merely records occasional download progress reports. In both cases, the exact size depends on the client's activity.

To quantify how much log data is generated, we ran NetSession-Base and NetSession-RCA with log uploading disabled; thus, each of 500 clients kept its *entire* 30-day log on its local disk. We then examined the log sizes at the end of the experiment. The average log size was 2.5 MB; the 5th and 95th percentiles were at 1.4 MB and 3.4 MB, respectively. The largest log was 86 MB. If we (conservatively) estimate the average download activity per client at 1 GB per month, a larger deployment with 100 million GUIDs would generate about 1.8 TB of logs per day. This corresponds to only 18 kB of logs per client per day.

Figure 4 provides a detailed breakdown of the log contents collected from the clients. A comparison with Figure 3 shows that the log is considerably smaller than a complete message trace; this is because a) the log contains only a hash of each data block rather than the actual bytes (which are known to the infrastructure anyway), and b) the log does not contain every single authenticator, but only the most recent one for each client.

Each client periodically uploads its log to the control plane and then deletes the entries once they have been acknowledged. Thus, the amount of storage that is needed locally on each client depends on the upload interval.

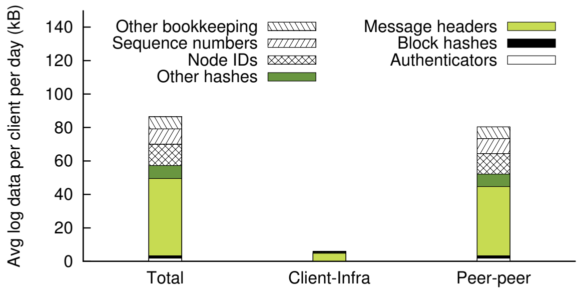


Figure 4: Average log size per client

With daily uploads, less than a MB of storage is required. Since every logged byte must eventually be uploaded, the amount of network traffic generated by the uploads is largely independent of the upload interval.

7.6 Cost: Log processing

Once the logs have been uploaded, the infrastructure must perform the consistency and plausibility checks described in Section 6. The required processing time depends on what actions are recorded in each log, but we expect it to be correlated with the overall log size.

To estimate the overhead, we performed the consistency and invariant checks on each of the logs produced by the clients. We measured the total processing time, as well as the fraction of time spent on consistency and invariant checks. Since we expect cryptographic operations to be a major factor, we separately measured the time spent verifying signatures (recall that the infrastructure does not generate signatures).

On average, about 0.78 MB's worth of log data was processed per second on a single CPU. 9% of the time was spent on consistency checking and 91% on invariant checking; overall, signature verifications accounted for about 3% of the processing time. Log processing can easily be parallelized, e.g., using MapReduce.

Based on these results, we estimate that a deployment with 100 million GUIDs would require about 28 extra machines to process the logs. For comparison, NetSession's current log processing system requires around 10 machines. Both estimates assume that the machines are fully utilized; in practice, log processing is one of several jobs that runs on a larger cluster.

7.7 Examples of statistical tests

Developing a full set of statistical tests for NetSession would require a detailed characterization of its workload, which is beyond the scope of the present paper. However, we use the set of simple tests in Table 2 to illustrate the general principle. These tests are fully automated; they are designed to constrain clients who collude to over-report uploads or deliver bad service to peers. We expect that more sophisticated tests will be based on de-

#	Which peers are quarantined?	Parameters	+Load
T1	IP has downloaded $> N_1$ bytes during the last k_1 days	$k_1 = 20$ $N_1 = 15.2G$	1.00%
T2	IP has been used by $> g_2$ GUIDs during the last k_2 days	$k_2 = 1$ $g_2 = 2$	0.57%
T3	GUID has downloaded $> N_3$ bytes during the last k_3 days	$k_3 = 20$ $N_3 = 10.4G$	0.99%
T4	GUID has downloaded $> N_4$ files during the last k_4 days	$k_4 = 20$ $N_4 = 140$	0.92%
T5	GUID failed to validate $> N_5$ bytes during the last k_5 days	$k_5 = 1$ $N_5 = 200K$	1.00%
	Tests T1–T5 combined		2.81%

Table 2: Statistical tests used with NetSession-RCA, along with the additional load (#bytes served) they place on the infrastructure due to quarantined clients.

tailed workload characteristics, e.g., channel switching patterns [11] or the clients' response to the quarantine.

More aggressive tests reduce the amount of misbehavior an adversary can get away with, but they also increase the load on the infrastructure due to false positives. To give a rough impression of how aggressive our simple tests could be, we used the 12/2010 trace to determine, for each test, the set of parameters that a) causes at most 1% additional load on the infrastructure, and among those, the one that b) constrains the adversary the most. These parameters, and the resulting load increases, are also shown in Table 2. Note that a given client can trigger more than one test; hence, the load increase from a set of tests is lower than the sum of the increases from the individual tests.

7.8 Effectivity

As a sanity check for the NetSession-RCA implementation, we injected a set of sample attacks. Specifically, we injected five inflation attacks and one corruption attack: In *blatant liars*, one client uploaded a fabricated log claiming to have downloaded 1 TB. In *collusion*, two clients continuously requested files and then reported that they had downloaded them from each other, regardless of which clients the infrastructure suggested. In *flash mob*, five clients joined the system simultaneously and rapidly requested rare files, 'faking' downloads whenever the infrastructure paired up two of them. In *leechers*, five clients joined the system simultaneously and downloaded random files as quickly as possible. In *Sybil attack*, one client joined the system with five GUIDs; the first GUID downloaded a rare file, and the others then tried to download the same file from each other. Finally, in *confused clients*, one client uploaded malformed log entries.

In all cases, the system behaved as expected. Logs from faulty clients were discarded, clients with abnormal behavior were quarantined, and the uploads affected by flash mob and Sybil attackers were effectively capped. In particular, *blatant liars* and *confused clients* were caught by our consistency checks (section 6.5), *collusion* was detected by the plausibility checks (section 6.6), *flash mob* was identified by resource certificates (section 6.3), and finally *leechers* and *Sybil attack* were flagged by statistical checks (section 6.7).

8 Statistical tests

As discussed in Section 6.7, RCA uses a set of statistical tests to decide which clients should be quarantined. We suggest the following general approach to choosing a suitable set of tests:

1. Identify the metrics that an attack is likely to affect;
2. Closely characterize the system's normal workload in those metrics; and
3. Choose a threshold for each metric such that, under the normal workload, no more than a small fraction of the clients are above the threshold (and would thus be quarantined).

A detailed characterization of NetSession's workload is beyond the scope of this paper, and is part of our ongoing work. However, for completeness we briefly summarize some of results from our initial investigation below.

One key set of metrics for NetSession is, obviously, the number of downloads and their distribution across the different files. Overall, file popularity in NetSession follows the usual power law, and the workload has the usual diurnal pattern. However, there is a lot more fine structure. For instance, some files are more popular at certain times of the day, and this pattern can vary between content providers; also, load can shift between files in a predictable pattern. To avoid being quarantined, an attacker would have to 'blend in' and closely imitate the current request pattern. This would be difficult, however, because of the information asymmetry that is inherent in hybrid systems like NetSession: the pattern is trivial to observe for the infrastructure but can only be approximated by an attacker.

Another set of metrics is related to the location of the clients, which can be obtained from a geolocation service such as Akamai's EdgeScape. The popularity of content can vary between regions, and it would be anomalous if content that is usually popular in the Middle East were to suddenly become popular in South America. Location-based metrics are particularly interesting because attackers cannot easily choose the location of compromised clients. Even botnets, a potential source of compromised clients, are often biased towards particular regions [32].

A third set of metrics is related to the download topology. Consider a graph with a vertex for each client and an edge for each pair of clients that have downloaded from one another. In this graph, which can easily be constructed by the control plane, a set of colluding peers would show up as a tightly connected subgraph. Efficient heuristics for detecting such subgraphs have already been developed for botnet detection, e.g., in BotGrep [26]. The control plane could use such techniques to detect suspicious subgraphs of clients, and then quarantine these clients or redirect their download requests to other, unrelated peers.

We note that the above list is not meant to be exhaustive or universal. A rich literature of other anomaly detection techniques exists, which could be applied to NetSession. On the other hand, the specific methods mentioned above may not be appropriate for all hybrid systems.

9 Conclusion

In this paper, we have examined a fundamental challenge in P2P-infrastructure hybrids: how to reliably account for the actions of untrusted clients. In current hybrid systems, malicious peers can report fictitious content downloads and degrade the system's quality of service. We described and evaluated RCA, a system that leverages the unique characteristics of P2P-infrastructure hybrids to limit the loss of accounting accuracy and service quality resulting from faulty or malicious clients. RCA reliably discovers all misreporting and protocol violations by individual clients, and it can automatically quarantine potentially colluding clients, at a moderate cost in terms of bandwidth and load on the infrastructure.

Acknowledgments

We thank our shepherd John Byers and the anonymous reviewers for their comments and suggestions. We also thank Miroslav Ponec and James Chalfant from Akamai Technologies for helpful comments on earlier drafts of this paper. This work was supported in part by NSF grants CNS-1040672 and CNS-1054229 and by the Max Planck Society.

References

[1] Joost. `http://www.joost.com/`.

[2] Limelight networks. `http://www.limelight.com/`.

[3] Octoshape. `http://www.octoshape.com/`.

[4] Pando networks. `http://www.pandonetworks.com/`.

[5] Velocix P2P assisted delivery. `http://www.velocix.com/network_delivery.php`.

[6] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. P. Wattenhofer. FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. In *Proc. OSDI*, Dec. 2002.

[7] S. Agarwal and J. R. Lorch. Matchmaking for online games and other latency-sensitive P2P systems. In *Proc. SIGCOMM*, 2009.

[8] Akamai acquires Red Swoosh. `http://www.akamai.com/html/about/press/releases/2007/press_041207.html`, Apr. 2007.

[9] M. Barreno, B. Nelson, R. Sears, A. D. Joseph, and J. D. Tygar. Can machine learning be secure? In *Proc. AsiaCCS*, 2006.

[10] R. A. Bazzi and G. Konjevod. On the establishment of distinct identities in overlay networks. In *PODC*, pages 312–320, 2005.

[11] M. Cha, P. Rodriguez, J. Crowcroft, S. Moon, and X. Amatriain. Watching television over an IP network. In *Proc. IMC*, 2008.

[12] E. Chan-Tin, D. Feldman, Y. Kim, and N. Hopper. The frog-boiling attack: Limitations of anomaly detection for secure network coordinates. In *Proc. SecureComm*, 2009.

[13] V. Chandola, A. Banerjee, and V. Kumar. Anomaly detection: A survey. *ACM Comput. Surv.*, 41:15:1–15:58, July 2009.

[14] B. Cohen. Incentives build robustness in BitTorrent. In *Proc. P2PEcon*, June 2003.

[15] D. E. Denning. An intrusion-detection model. *IEEE Trans. on Software Engineering*, 13(2):222–232, 1987.

[16] M. Dischinger, A. Haeberlen, K. P. Gummadi, and S. Saroiu. Characterizing residential broadband networks. In *Proc. IMC*, Oct 2007.

[17] J. R. Douceur. The Sybil attack. In *Proc. IPTPS*, Mar 2002.

[18] M. J. Freedman and R. Morris. Tarzan: a peer-to-peer anonymizing network layer. In *Proc. ACM CCS*, 2002.

[19] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: A virtual machine-based platform for trusted computing. In *Proc. SOSP*, Oct. 2003.

[20] A. Haeberlen, P. Kuznetsov, and P. Druschel. PeerReview: Practical accountability for distributed systems. In *SOSP*, Oct. 2007.

[21] C. Huang, A. Wang, J. Li, and K. W. Ross. Understanding hybrid CDN-P2P: why Limelight needs its own Red Swoosh. In *Proc. NOSSDAV*, 2008.

[22] T. Karagiannis, P. Rodriguez, and K. Papagiannaki. Should Internet service providers fear peer-assisted content distribution? In *Proc. IMC*, 2005.

[23] D. M. Lewin, B. Maggs, and J. J. Kloninger. Internet Content Delivery Service with Third Party Cache Interface Support. U.S. Patent Number 7,010,578, Mar. 2006.

[24] Q. Lian, Z. Zhang, M. Yang, B. Y. Zhao, Y. Dai, and X. Li. An empirical study of collusion behavior in the Maze P2P file-sharing system. In *Proc. ICDCS*, 2007.

[25] N. B. Margolin and B. N. Levine. Financial cryptography and data security; Chapter "Quantifying resistance to the Sybil attack". Springer-Verlag, 2008.

[26] S. Nagaraja, P. Mittal, C.-Y. Hong, M. Caesar, and N. Borisov. Botgrep: finding P2P bots with structured graph analysis. In *Proceedings of the 19th USENIX conference on Security*, Aug. 2010.

[27] B. Parno, D. Wendlandt, E. Shi, A. Perrig, B. M. Maggs, and Y.-C. Hu. Portcullis: Protecting connection setup from denial-of-capability attacks. In *Proc. SIGCOMM*, 2007.

[28] R. S. Peterson and E. G. Sirer. Antfarm: efficient content distribution with managed swarms. In *Proc. NSDI*, 2009.

[29] Major features in PunkBuster. `http://www.evenbalance.com/index.php?page=info.php`.

[30] S. Seuken and D. C. Parkes. On the Sybil-proofness of accounting mechanisms. In *Proc. NetEcon*, June 2011.

[31] M. Sirivianos, J. H. Park, X. Yang, and S. Jarecki. Dandelion: Cooperative content distribution with robust incentives. In *Proc. USENIX ATC*, June 2007.

[32] B. Stone-Gross, M. Cova, L. Cavallaro, B. Gilbert, M. Szydlowski, R. Kemmerer, C. Kruegel, and G. Vigna. Your botnet is my botnet: analysis of a botnet takeover. In *Proc. CCS*, 2009.

[33] L. Vu, I. Gupta, K. Nahrstedt, and J. Liang. Understanding overlay characteristics of a large-scale peer-to-peer IPTV system. *ACM Trans. Multim. Comp. Comm. Appl.*, 6:31:1–31:24, 2010.

[34] C. Wu, B. Li, and S. Zhao. Diagnosing network-wide P2P live streaming inefficiencies. In *Proc. IEEE INFOCOM*, Apr. 2009.

[35] H. Yin, X. Liu, T. Zhan, V. Sekar, F. Qiu, C. Lin, H. Zhang, and B. Li. Design and deployment of a hybrid CDN-P2P system for live video streaming: experiences with LiveSky. In *Proc. ACM MM*, 2009.

[36] H. Yu, P. B. Gibbons, M. Kaminsky, and F. Xiao. SybilLimit: A near-optimal social network defense against Sybil attacks. In *Proc. IEEE S&P*, 2008.

[37] H. Yu, M. Kaminsky, P. B. Gibbons, and A. Flaxman. Sybilguard: defending against Sybil attacks via social networks. In *Proc. SIGCOMM '06*, Aug. 2006.

Header Space Analysis: Static Checking For Networks

Peyman Kazemian
Stanford University
kazemian@stanford.edu

George Varghese
UCSD and Yahoo! Research
varghese@cs.ucsd.edu

Nick McKeown
Stanford University
nickm@stanford.edu

Abstract

Today's networks typically carry or deploy dozens of protocols and mechanisms simultaneously such as MPLS, NAT, ACLs and route redistribution. Even when individual protocols function correctly, failures can arise from the complex interactions of their aggregate, requiring network administrators to be masters of detail. Our goal is to automatically find an important class of failures, regardless of the protocols running, for both operational and experimental networks.

To this end we developed a general and protocol-agnostic framework, called *Header Space Analysis* (HSA). Our formalism allows us to statically check network specifications and configurations to identify an important class of failures such as *Reachability* Failures, *Forwarding Loops* and *Traffic Isolation and Leakage* problems. In HSA, protocol header fields are not first class entities; instead we look at the entire packet header as a concatenation of bits without any associated meaning. Each packet is a point in the $\{0,1\}^L$ space where L is the maximum length of a packet header, and networking boxes transform packets from one point in the space to another point or set of points (multicast).

We created a library of tools, called Hassel, to implement our framework, and used it to analyze a variety of networks and protocols. Hassel was used to analyze the Stanford University backbone network, and found all the forwarding loops in less than 10 minutes, and verified reachability constraints between two subnets in 13 seconds. It also found a large and complex loop in an experimental loose source routing protocol in 4 minutes.

1 Introduction

"Accidents will occur in the best-regulated families" — *Charles Dickens*

In the beginning, a switch or router was breathtakingly simple. About all the device needed to do was index into a forwarding table using a destination address, and decide where to send the packet next. Over time, forwarding grew more complicated. Middleboxes (e.g., NAT and firewalls) and encapsulation mechanisms (e.g., VLAN and MPLS) appeared to escape from IP's limitations: e.g., NAT bypasses address limits and MPLS allows flexible routing. Further, new protocols for specific domains, such as data centers, WANs and wireless, have greatly increased the complexity of packet forwarding. Today, there are over 6,000 Internet RFCs and it is not unusual for a switch or router to handle ten or more encapsulation formats simultaneously.

This complexity makes it daunting to operate a large network today. Network operators require great sophistication to master the complexity of many interacting protocols and middleboxes. The future is not any more rosy - complexity today makes operators wary of trying new protocols, even if they are available, for fear of breaking their network. Complexity also makes networks fragile, and susceptible to problems where hosts become isolated and unable to communicate. Debugging reachability problems is very time consuming. Even simple questions are hard to answer, such as *"Can Host A talk to Host B?"* or *"Can packets loop in my network?"* or *"Can User A listen to communications between Users B and C?"*. These questions are especially hard to answer in networks carrying multiple encapsulations and containing boxes that filter packets.

Thus, our first goal is to help system administrators statically analyze production networks today. We describe new methods and tools to provide formal answers to these questions, and many other failure conditions, *regardless of the protocols running in the network.*

Our second goal is to make it easier for system administrators to guarantee isolation between sets of hosts, users or traffic. Partitioning networks this way is usually called "slicing"; VLANs are a simple example used today. If configured correctly, we can be confident that traffic in one slice (e.g. a VLAN) cannot leak into another. This is useful for security, and to help answer questions such as *"Can I prevent Host A from talking to Host B?"*. For example, imagine two health-care providers using the same physical network. HIPAA [20] rules require that no information about a patient can be read by other providers. Thus a natural application of slicing is to place each provider in a separate slice and guarantee that no packet from one slice can be controlled by or read by the other slice. We call this *secure slicing*. Secure slicing may also be useful for banks as part of defense-in-depth, and for classified and unclassified users sharing the same physical network. Our tools can verify that slices have been correctly configured.

Our third goal is to take the notion of isolation further, and enable the static analysis of networks sliced in more general ways. For example, with FlowVisor [6] a slice can be defined by any combination of header fields. A slice consists of a topology of switches and links, the set of headers on each link, and its share of link capacity. Each slice has its own control plane, allowing its owner to decide how packets are routed and processed. While tools such as FlowVisor allow rapid deployment of new protocols, they add to the complexity of the network, pushing the level of detail beyond the comprehension of a human operator. Our tools allow automatic analysis of the network configuration to formally prove that the slicing is operating as intended.

In the face of this need, it is surprising that there are very few existing network management tools to analyze large networks. Further, the tools that exist are protocol dependent and specialized to each task. For example, the pioneering work of Xie, *et al* [4] on static reachability analysis, the analyses of IP connectivity and firewall configuration, e.g. [11, 9, 10, 15], and work on routing failures [12, 13] are all tailored to IP networks. While these papers suggest powerful approaches for reachability in IP networks, they do not easily extend to new protocols and new types of checks.

This paper introduces a general framework, called *Header Space Analysis*, which provides a set of tools and insights to model and check networks for a variety of failure conditions in a protocol-independent way. Key to our approach is a generalization of the geometric approach to packet classification pioneered by Lakshman and Stiliadis [3], in which classification rules over K packet fields are viewed as subspaces in a K dimensional space.

We generalize in three ways. First, we jettison the notion of pre-specified fields in favor of a header space of L bits where each packet is represented by a point in $\{0, 1\}^L$ space, where L is the header length. This allows us to work with emerging protocols and arbitrary field formats. Second, we go beyond modeling packet classification in which a header is mapped to a single point in a matching subspace. Instead, we model *all* router and middlebox processing as *box transfer functions* transforming subspaces of the L-dimensional space to other subspaces. For example, in Figure 1, A and B are arbitrary boxes, and T_A and T_B represent their transfer functions. We model how a packet or flow is modified as it travels by composing the transfer functions along the path. Third, we go beyond modeling a single box to modeling a network of boxes using a *network transfer function,* Ψ and a *topology transfer function,* Γ. Ψ combines all individual box functions into one giant function. Γ models the links that connect ports together. The overall behavior of the network is modeled as a black box by composing Ψ and Γ along all paths.

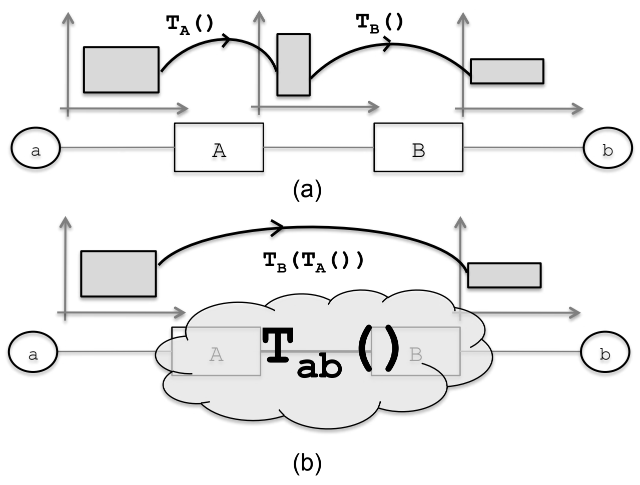


Figure 1: (a) Changes to a flow as it passes through two boxes with transfer function T_A and T_B. (b) Composing transfer functions to model end to end behavior of a network.

The contributions of this paper and an outline of the rest of the paper are as follows:

- *Header Space Analysis:* Section 2 describes the geometric model and defines transfer functions. Section 3 shows how transfer functions can be used to model today's networking boxes. Section 4 describes an algebra for working on header space.
- *Use Cases:* Section 5 describes how header space analysis can be used to detect network failures such as reachability failures, routing loops and slice isolation in a protocol independent way.
- *Implementation:* Section 6 describes a library of tools (called *Hassel, or Header Space Library*), based on header space analysis, that can statically analyze networks. We describe five key optimizations that boost Hassel's performance by 5 orders of magnitude relative to a naive implementation.
- *Experiments:* Section 7 reports results of using Hassel to analyze three examples: (1) Stanford University's backbone network, (2) Slice isolation check, and (3) An experimental source routing protocol. We report loops found, and show that even our Python implementation scales to large enterprise networks with our optimizations.

We describe limitations of our approach and related work in Sections 8 and 9. We conclude in Section 10.

2 The Geometric Model

Our *header space* framework is built on a geometric model. We model packets as points *in* a geometric space and network boxes as transfer functions *on* the same geometric space. Our first task is to define the main geometric spaces.

Header Space, $\mathcal{H}$: We ignore the protocol-specific meanings associated with header bits and view a packet header as a flat sequence of ones and zeros. Formally, a header is a point and a flow is a region in the $\{0,1\}^L$ space, where L is an upper bound on the header length. We call this space *Header Space, $\mathcal{H}$*.

A wildcard expression is the basic building block used to define objects in $\mathcal{H}$. Each wildcard expression is a sequence of L bits where each bit can be either 0, 1 or x. Each wildcard expression corresponds to a hypercube in $\mathcal{H}$. Every region, or flow, in $\mathcal{H}$ is defined as a *union* of wildcard expressions.

$\mathcal{H}$ abstracts away the data portion of a packet because we assume it does not affect packet processing. If it does, as in an intrusion-detection box, then L must be the length of the entire packet. If the fields are fixed, we can define macros for each field in $\mathcal{H}$ to reduce dimensionality. However, the general notion of header space is critical when dealing with different protocols that interpret the same header bits in different ways. Note also that we can model variable length fields such as IP options using a custom parsing function as shown in Section 7.3.

Network Space, $\mathcal{N}$: We model the network as a set of boxes called *switches* with external interfaces called *ports* each of which is modeled as having a unique identifier. We use "switches" to denote routers, bridges, and any possible middlebox.

If we take the cross-product of the switch-port space (the space of all ports in the network, $\mathcal{S}$) with $\mathcal{H}$, we can represent a packet traversing on a link as a point in $\{0,1\}^L \times \{1,...,P\}$ space, where $\{1,...,P\}$ is the list of ports in the network. We call the space of all possible packet headers, localized at all possible input ports in the network, the *Network Space*, $\mathcal{N}$.

Network Transfer Function, $\Psi()$: As a packet traverses the network, it is transformed from one point in Network Space to other point(s) in Network Space. For example, a layer 2 switch, that merely forwards a packet from one port to another, without rewriting headers, transforms packets only along the switch-port axis, $\mathcal{S}$. On the other hand, an IPv4 router that rewrites some fields (e.g. MAC address, TTL, checksum) and then forwards the packet, transforms the packet both in $\mathcal{H}$ and $\mathcal{S}$.

As these examples suggest, all networking boxes can be modeled as *Transformers* with a *Transfer Function* (see Figure 1), that models their protocol dependent functions. More precisely, a node can be modeled using its transfer function, T, that maps header h arriving on port p:

$$T(h,p): \quad (h,p) \rightarrow \{(h_1,p_1),(h_2,p_2),...\}$$

In general, the transfer function may depend on the input port to model input-port-specific behavior and the output may be a set of $(header, port)$ pairs to allow multicasting.[1]

A concept we use heavily is the *network transfer function*, $\Psi(.)$. Given that switch ports are numbered uniquely, we combine all the box transfer functions into a composite transfer function describing the overall behavior of the network. Formally, if a network consists of n boxes with transfer functions $T_1(.),...,T_n(.)$, then:

$$\Psi(h,p) = \begin{cases} T_1(h,p) & \text{if } p \in switch_1 \\ ... & ... \\ T_n(h,p) & \text{if } p \in switch_n \end{cases}$$

Topology Transfer Function, $\Gamma()$: We can model the network topology using a *topology transfer function*, $\Gamma()$, defined as:

$$\Gamma(h,p) = \begin{cases} \{(h,p^*)\} & \text{if } p \text{ connected to } p^* \\ \{\} & \text{if } p \text{ is not connected.} \end{cases}$$

Γ models the behavior of links in the network. It accepts a packet at one end of a link and returns the same packet, unchanged, at the other end. Note that links are unidirectional in this model. To model bidirectional links, one rule should be added per direction.

Multihop Packet Traversal: Using the two transfer functions, we can model a packet as it traverses the network by applying $\Phi(.) = \Psi(\Gamma(.))$ at each hop. For example, if a packet with header h enters a network on port p, the header after k hops will be $\Psi(\Gamma(...(\Psi(\Gamma(h,p)...)$, or simply $\Phi^k(h,p)$: each Γ forwards the packet on a link and each Ψ passes the packet through a box.

Slice: A slice, S, can be defined as (*Slice network space*, *Permission*, *Slice Transfer Function*) where *Slice network space* is a subset of the network space controlled by the slice, and Permission is a subset of {read(r), write(w)}[2]. The Slice Transfer Function, $\Psi_s(h,p)$, captures the behavior of all rules installed by the control plane of slice S. For example, a slice that controls packets destined to subnet 192.168.1.0/24 and is restricted to network ports 1, 2 and 3 can be expressed as (($ip\_dst(h)$ = 192.168.1.x , $p \in \{1,2,3\}$), rw , Ψ_s). Here, $ip\_dst(h)$ is a helper function refering to the IP destination bits in the header.

Our concept of a slice combines two notions we normally think of as very different. It describes the *implicit* slicing, when protocols coexist today on the same network using protocol IDs (e.g. TCP and UDP) or networks partitioned using Vlan IDs. It also describes the

[1]It also enables us to model load balancing boxes for which the output port is a psuedo-random function of the header bits.

[2]A real slice may have other attributes such as bandwidth reservations, but our model ignores attributes irrelevant to the analysis.

explicit slicing utilized by FlowVisor [6] to create independent experiments in an OpenFlow [5] network, as done in testbed networks such as GENI[3] [19].

3 Modeling Networking Boxes

This section is a brief tutorial on transfer functions in order to illustrate their power in modeling different boxes in a unified way. We use helper functions for clarity. We refer to a particular field in a particular protocol using helper function $protocol\_field()$. For example, $ip\_src(h)$ refers to the source IP address bits of header h. Similarly, helper function $\mathcal{R}(h, fields, values)$ is used to rewrite the *fields* in h with *values*. For example, $\mathcal{R}(h, mac\_dst(), d)$ rewrites the MAC destination address to d. Header updates can be represented by a *masking AND* followed by a *rewrite OR*.

We start by modeling an IPv4 router which processes packets as follows: 1) Rewrite source and destination MAC addresses, 2) Decrement TTL, 3) Update checksum, 4) Forward to outgoing port. Thus the transfer function of an IPv4 router concatenates four functions:

$$T_{IPv4}(.) = T_{fwd}(T_{chksum}(T_{ttl}(T_{mac}(.)))).$$

We examine each function in turn. $T_{fwd}(.)$ looks up $ip\_dst(h)$ in a lookup table and returns the output port. If lookup is modeled as $ip\_lookup(.) : ip\_dst \rightarrow port$:

$$T_{fwd}(h, p) = \{(h, ip\_lookup(ip\_dst(h)))\}$$

Similarly, $T_{mac}(.)$ looks up the next hop MAC address and updates source and destination MAC addresses. $T_{ttl}(.)$ drops the packet if $ip\_ttl(h)$ is 0 and otherwise does $\mathcal{R}(h, ip\_ttl(), ip\_ttl(h) - 1)$. $T_{chksum}(.)$ updates the IP checksum. If the focus is on IP routing, we might choose to ignore $T_{mac}(.)$, simplifying the model to $T_{IPv4}(.) = T_{fwd}(T_{ttl}(.))$ or even $T_{IPv4}(.) = T_{fwd}()$. As an example, a simplified transfer function of an IPv4 router that forwards subnet S_1 traffic to port p_1, S_2 traffic to port p_2 and S_3 traffic to port p_3 is:

$$T_r(h,p) = \begin{cases} \{(h, p_1)\} & \text{if } ip\_dst(h) \in S_1 \\ \{(h, p_2)\} & \text{if } ip\_dst(h) \in S_2 \\ \{(h, p_3)\} & \text{if } ip\_dst(h) \in S_3 \\ \{\} & \textit{otherwise.} \end{cases}$$

A firewall is modeled as a transfer function that extracts IP and TCP headers, matches the headers against a sequence of wildcard expressions (which model ACL rules), and drops or forwards the packet as specified by the matching rule. A tunneling end point is modeled using a shift operator that shifts the payload packet to the right and a rewrite operator that rewrites the beginning of the header. A Network Address Translator (NAT) box can also be modeled using a rewrite operator. However the level of details that we use in our model depends on the application. For example we can have a detailed model where we model the exact source IP to source transport port mapping, or we may set the output transport source port to a wildcard (all x) to represent every possible mapping. In [1] we provide more examples. While modeling is trivial but tedious, we have written tools that parse router configuration files and forwarding tables to automate the process.

[3] In GENI parlance, the network space of a slice is called *flow space.*

4 Header Space Algebra

Algorithms that compute reachability or determine if two slices can interact must determine how different spaces overlap. We therefore need to define basic set operations on $\mathcal{H}$: *intersection, union, complementation* and *difference*. We also define the *Domain, Range* and *Range Inverse* for transfer functions. The next section shows how this algebra is used.

4.1 Set Operations on $\mathcal{H}$

While set operations on bit vectors are well-known, we need set operations on *wildcard* expressions. Since all objects in header space can be represented as a union of wildcard expressions, defining set operation on wildcard expressions allows these operations to carry over to header space objects. For the rest of this paper, we overload the term *header* to refer to both packet headers (points in $\mathcal{H}$) and wildcard expressions (hyper-cubes in $\mathcal{H}$).

Intersection: For two headers to have a non-empty intersection, both headers *must* have the same bit value at every position that is not a wildcard. If two headers differ in bit b_i, then the two headers will be in different hyper-planes defined by $b_i = 0$ and $b_i = 1$. On the other hand, if one header has an x in a position while the other header has a 1 or 0, the intersection is non-empty. Thus, the *single-bit intersection* rule for $b_i \cap b'_i$ is defined as:

b_i \ b'_i	0	1	x
0	0	z	0
1	z	1	1
x	0	1	x

In the table, z means the bitwise intersection is empty. The intersection of two headers is found by applying the single-bit intersection rule, bit-by-bit, to the headers. z is an "annihilator": if any bit returns z, the intersection of all bits is empty. As an example, 11000xxx $\cap$ xx00010x = 1100010x and 1100xxxx $\cap$ 111001xx = 11z001xx = ϕ. A simple trick allows efficient software implementation.

Encode each bit in the header using *two* bits: $0 \rightarrow 01$, $1 \rightarrow 10$, $x \rightarrow 11$ and $z \rightarrow 00$. Intersection, then, is simply an AND operation on the encoded headers.

Union: In general, a union of wildcard expressions cannot be simplified. For example, no single header can represent the union of 1111xxxx and 0000xxxx. This is why a header space object is defined as a union of wildcard expressions. In some cases, we can simplify the union (e.g., 1100xxxx ∪ 1000xxxx simplifies to 1x00xxxx) by simplifying an equivalent boolean expression. For example, 10xx ∪ 011x is equivalent to $b_4\overline{b_3} \oplus \overline{b_4}b_3b_2$. This allows the use of Karnaugh Maps and Quine-McCluskey [18] algorithms for logic minimization.

Complementation: The complement of header h — the union of all headers that do *not* intersect with h — is computed as follows:

```
$h' \leftarrow \phi$
**for** bit $b_i$ in $h$ **do**
    **if** $b_i \neq x$ **then**
        $h' \leftarrow h' \cup$ x...x$\overline{b_i}$x...x
    **end if**
**end for**
**return** $h'$
```

The algorithm finds all non-intersecting headers by replacing each 0 or 1 in the header with its complement. This follows because just one non-intersecting bit (or z) in a term results in a disjoint header. For example, (100xxxxx)′ = 0xxxxxxx ∪ x1xxxxxx ∪ xx1xxxxx.

Difference: The difference (or minus) operation can be calculated using intersection and complementation. $A - B = A \cap B'$. For example:

$$\begin{aligned}&100\text{xxxxx} - 10011\text{xxx} =\\&100\text{xxxxx} \cap (0\text{xxxxxxx} \cup \text{x1xxxxxx} \cup \text{xx1xxxxx}\\&\cup \text{xxx0xxxx} \cup \text{xxxx0xxx})\\&= \phi \cup \phi \cup \phi \cup 1000\text{xxxx} \cup 100\text{x0xxx}\\&= 1000\text{xxxx} \cup 100\text{x0xxx}.\end{aligned}$$

The difference operation can be used to check if one header is a subset of another: $A \subseteq B \iff A - B = \phi$.

4.2 Domain, Range and Range Inverse

To capture the destiny of packets through a box or set of boxes, we define the *domain*, *range* and *range inverse* as follows:

Domain: The domain of a transfer function is the set of all possible *(header, port)* pairs that the transfer function accepts. Even headers for which the output is empty (i.e., dropped packets) belong to the domain.

Range: The range of a transfer function is the set of all possible *(header, port)* pairs that the transfer function can output after applying all possible inputs on every port.

Range Inverse: Reachability and loop detection computation requires working backwards from a range to determine what input *(header, port)* pairs could have produced it. If $S = \{(h_1, p_1), ..., (h_j, p_2)\}$, then the range inverse of S under transfer function $T(.)$ is $X = \{(h_i, p_i)\}]_1^n$ such that $T(X) = S$. Equivalently, $X = T^{-1}(S)$. The inverse of a transfer function is well-defined: A transfer function maps each (h, p) pair to a set of other pairs. By following the mapping backward, we can invert a transfer function.

5 Using Header Space Analysis

In this section we show how the header space analysis – developed in the last three sections – can be used for solving several classical networking problems in a protocol-agnostic way.

5.1 Reachability Analysis

Xie, et. al. [4] analyze reachability by tracing which of all possible packet headers at a source can reach a destination. We follow a similar approach, but generalize to arbitrary protocols. Using header space analysis, we consider the space of all headers leaving the source, then track this space as it is transformed by each successive networking box along the path (or paths) to the destination. At the destination, if no header space remains, the two hosts cannot communicate. Otherwise, we trace the remained header spaces backwards (using the range inverse at each step) to find the set of headers the source can send to reach the destination.

Consider, for example, the question: *Can packets from host a reach host b?*. Define the reachability function R between a and b as:

$$R_{a\rightarrow b} = \bigcup_{a\rightarrow b \text{ paths}} \{T_n(\Gamma(T_{n-1}(......(\Gamma(T_1(h,p)...))\}$$

where for each path between a and b, $\{T_1, ..., T_{n-1}, T_n\}$ are the transfer functions along the path. The switches in each path are denoted by:

$$a \rightarrow S_1 \rightarrow ... \rightarrow S_{n-1} \rightarrow S_n \rightarrow b.$$

The *Range* of $R_{a\rightarrow b}$ is the set of headers that can reach b from a. Notice that these headers are *seen at* b, and not necessarily headers transmitted by a, since headers may change in transit. We can find which packet headers can leave a and reach b by computing the range inverse. If header $h \subset \mathcal{H}$ reached b along the $a \rightarrow S_1 \rightarrow ... \rightarrow S_{n-1} \rightarrow S_n \rightarrow b$ path, then the original header sent by a is:

$$h_a = T_1^{-1}(\Gamma(...(T_{n-1}^{-1}(\Gamma(T_n^{-1}((h,b))...)),$$

using the fact that $\Gamma = \Gamma^{-1}$.

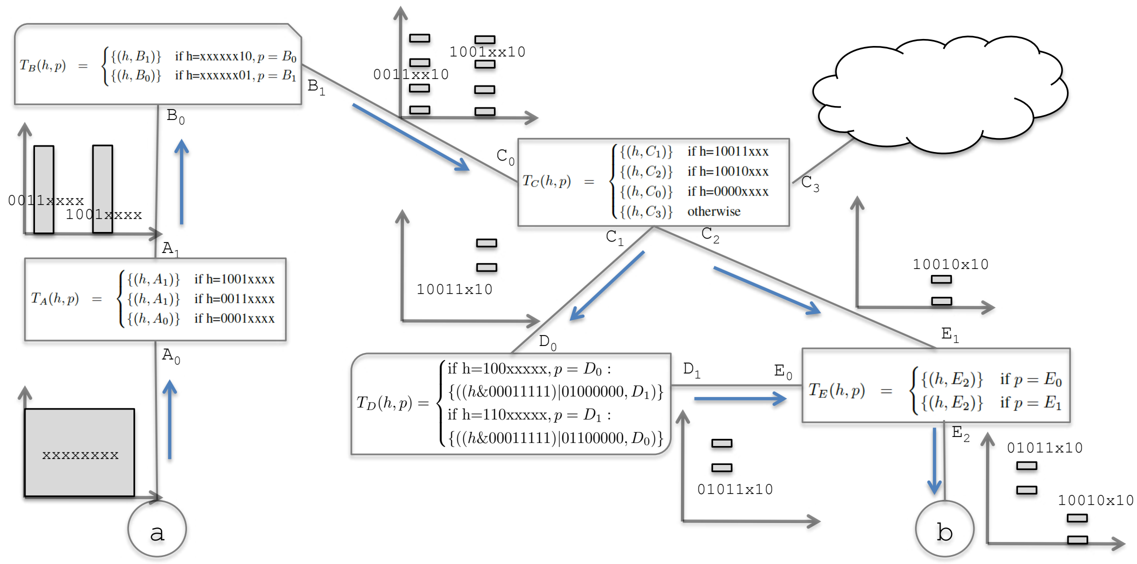

$T_B(h,p) = \begin{cases} \{(h,B_1)\} & \text{if h=xxxxxx10}, p=B_0 \\ \{(h,B_0)\} & \text{if h=xxxxxx01}, p=B_1 \end{cases}$
B1
B0
0011xxxx
1001xxxx
A1
$T_A(h,p) = \begin{cases} \{(h,A_1)\} & \text{if h=1001xxxx} \\ \{(h,A_1)\} & \text{if h=0011xxxx} \\ \{(h,A_0)\} & \text{if h=0001xxxx} \end{cases}$
A0
xxxxxxxx
a
0011xx10
1001xx10
C0
$T_C(h,p) = \begin{cases} \{(h,C_1)\} & \text{if h=10011xxx} \\ \{(h,C_2)\} & \text{if h=10010xxx} \\ \{(h,C_0)\} & \text{if h=0000xxxx} \\ \{(h,C_3)\} & \text{otherwise} \end{cases}$
C3
C1
C2
10011x10
10010x10
D0
$T_D(h,p) = \begin{cases} \text{if h=100xxxxx}, p=D_0: \\ \{((h\&00011111)|01000000, D_1)\} \\ \text{if h=110xxxxx}, p=D_1: \\ \{((h\&00011111)|01100000, D_0)\} \end{cases}$
D1
E0
E1
$T_E(h,p) = \begin{cases} \{(h,E_2)\} & \text{if } p=E_0 \\ \{(h,E_2)\} & \text{if } p=E_1 \end{cases}$
E2
01011x10
01011x10
10010x10
b

Figure 2: Example for computing reachability function from a to b. For simplicity, we assume a header length of 8 and show the first 4 bits on the x-axis and the last 4 bits on the y-axis. We show the range (output) of each transfer function composition along the paths that connect a to b. At the end, the packet headers that b will see from a are 01011x10 $\cup$ 10010x10.

To provide intuition, we do reachability analysis for the small example network in Figure 2. Each box in Figure 2 contains its transfer function. To keep things simple, we only use 8-bit headers; since we cannot easily depict eight dimensions, we represent the first 4 bits of the header on the x-axis and the last 4 bits on the y-axis. Note that in this example, A and C are miniature models of IP routers, B is a firewall, D is a simplified NAT box and E behaves like an Ethernet switch.

Figure 2 shows how the network boxes transform header space along each path. By repeatedly applying the output of each transfer function to the input of the next transfer function in each path, the reachability function from a to b becomes:

$$R_{a\to b}(h,p) = \begin{cases} \text{if h=10010x10}, p = A_0: \\ \{(h, E_2)\} \\ \text{if h=10011x10}, p = A_0: \\ \{((h\&00011111)|01000000, E_2)\} \end{cases}$$

The range of $R_{a\to b}$, which is the final output set in Figure 2, is 10010x10 $\cup$ 01011x10. This is the set of headers that can reach b from a. To find the set of headers that a can send to b, we compute the range inverse of $R_{a\to b}$ which is 10010x10 $\cup$ 10011x10.

Complexity: As we push the test packet toward the destination, the transfer function rules divide the input headerspace into smaller pieces. If the headerspace consists of the union of R_1 wildcard expressions and the transfer function has R_2 rules, then the output can be a headerspace with $O(R_1R_2)$ wildcard expressions. However, this is the worst case scenario. In a real network whose purpose is to provide connectivity, as the header space propagates to the core of the network, the match patterns of forwarding rules will become less specific, therefore the space will not be divided into too many pieces. Most of the flow division happens as a result of rules that are filtering out some part of input flow (e.g. ACL rules). As a result, each of the input wildcard expressions will match only a few rules in the transfer function and generate at most cR (and not R^2), where $c << R$. We call this, the *Linear Fragmentation* assumption. Under this assumption, the running time is $O(dR^2)$ where d is the network diameter – the maximum number of hubs that a packet will go through before reaching the destination – and R is the maximum number of forwarding rules in a router. See [1] for more details. While our algorithm may appear almost to be a simulation, we gain algorithmic leverage by treating groups of headers as an equivalence class wherever possible. By contrast, a brute-force algorithm that simulates the sending of every possible packet has $O(2^L)$ complexity.

5.2 Loop Detection

A loop occurs when a packet returns to a port it has visited earlier. Header space analysis can determine all packet headers that loop. We first describe how to detect *generic* loops and then show how to detect *infinite* loops, a subset of generic loops where packets loop indefinitely. An example of a generic, but finite loop, is an IP packet that loops until the TTL decrements to zero.

Generic Loops: Given a network transfer function, we detect all loops by injecting an *all-x* test packet header

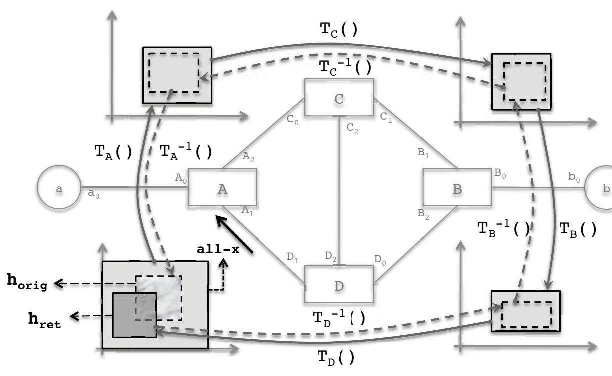
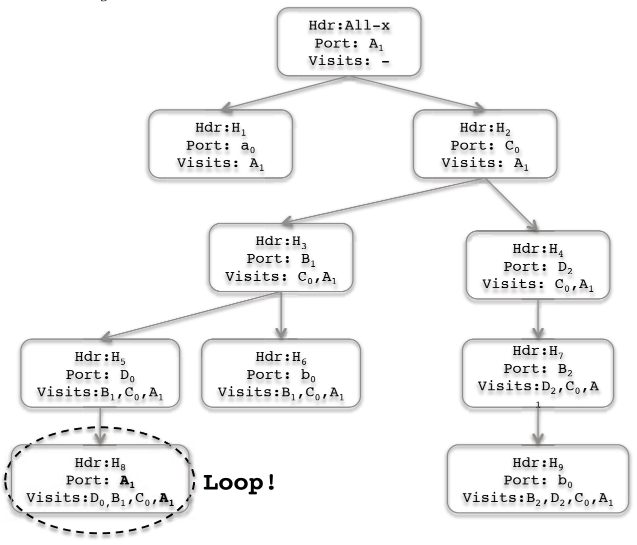


Figure 3: An example network for running the loop detection algorithm. The solid lines show the changes in the all-x test packet injected from A_1 till it returns to the injection port as h_{ret}. The dashed lines show the process of detecting infinite loop, where h_{ret} is traced back to find h_{orig}, the part of all-x packet that caused h_{ret}.



Hdr:All-x Port: A1 Visits: -
Hdr:H1 Port: a0 Visits: A1
Hdr:H2 Port: C0 Visits: A1
Hdr:H3 Port: B1 Visits: C0,A1
Hdr:H4 Port: D2 Visits: C0,A1
Hdr:H5 Port: D0 Visits:B1,C0,A1
Hdr:H6 Port: b0 Visits:B1,C0,A1
Hdr:H7 Port: B2 Visits:D2,C0,A1
Hdr:H8 Port: A1 Visits:D0,B1,C0,A1
Loop!
Hdr:H9 Port: b0 Visits:B2,D2,C0,A1

Figure 4: Example of the propagation graph for a test packet injected from port A_1 in network of figure 3.

(i.e., a packet header, all of whose bits are wild-carded) from *each port* in the network and track the packet until:

- (Case 1) It leaves the network;
- (Case 2) It returns to a *port* already visited (P_{ret}); or
- (Case 3) It returns to the port[4] it was injected from (P_{inj}).

Only in Case 3 – i.e. the packet comes back to its injection port — do we report a loop. Since we repeat the same procedure starting at every port, we will detect the same loop when we inject a test packet from P_{ret}. Ignoring Case 2 avoids reporting the same loop twice.

We find loops using *breadth first search* on the *propagation graph*. For example, in Figure 3 we inject the all-x test packet into port A_1. Figure 4 is the corresponding propagation graph. Each node in the propagation graph shows the *set* of packet headers, `Hdr`, that reached a `Port` and the set of ports visited previously in their path: `Visits`. For example, in Figure 4:

$$\{(H_3, B_1), (H_4, D_2)\} = \Phi(H_2, C_0).$$

We detect a loop when `Port` is the first element of `Visits`.[5]

The generic loop detection test has the same algorithmic structure as reachability test, and hence its complexity is similarly $O(dPR^2)$ under the *Linear Fragmentation* assumption. Here P is number of ports that we need to inject the test packet from. See [1] for more details.

Finding Single Infinite Loops: Not all generic loops are infinite. For example, in the loop "$A \to C \to B \to D \to A$" in Figure 3, the header space changes as the test packet traverses the loop. Let h_{ret} denote the part of header space that returns to A_1. Then h_{orig}, defined as

$$h_{orig} = \Phi^{-1}(\Phi^{-1}(\Phi^{-1}(\Phi^{-1}(h_{ret}, A_1))))$$

is the original header space that produces h_{ret}. Figure 3 also depicts the process of finding h_{orig}.

Now, h_{ret} and h_{orig} relate in one of three ways:

1. $h_{ret} \cap h_{orig} = \phi$: In this case, the loop is surely finite. The header space that caused the loop, i.e. h_{orig}, does not intersect with the returned header space, and the loop will terminate.

2. $h_{ret} \subseteq h_{orig}$: in this case the loop is certainly infinite. Every packet header in h_{orig} is mapped by the transfer function of the loop to a point in h_{ret}. Since h_{ret} is completely within h_{orig}, the process will repeat in the next round, and the loop will continue indefinitely.

3. Neither of the above: In this case, we need to iterate again. First, note that $h_{ret} - h_{orig}$ completely satisfies Case 1's condition, and therefore cannot loop again. But we must examine $h_{ret} \cap h_{orig}$. So we redefine $h_{ret} := h_{ret} \cap h_{orig}$ and calculate the new h_{orig}. We repeat until one of the first two cases happens. The process must terminate (in at most 2^L steps) because at each step the newly defined h_{ret} shrinks. Hence, eventually case 1 or 2 will happen, or h_{ret} will be empty.[6]

More tortuous loops, where a packet passes through other loops before coming back to a first loop, can also be detected using a simple generalization [1].

5.3 Slice Isolation

Network operators often wish to control which groups of hosts (or users) can communicate with each other.

[4]While we could define a loop as a packet returning to a *node* visited earlier, using ports helps detect infinite loops.

[5]If `Port` appears anywhere else in `Visits`, we terminate the branch.

[6]IP TTL is an example of Case 3. For a loop of length n, $h_{ret} = ttl$ for $0 < ttl < 256 - n$ and $h_{orig} = ttl$ for $n < ttl < 256$. In the next round $h_{ret} := h_{ret} \cap h_{orig} = ttl$ for $n < ttl < 256 - n$. In subsequent rounds h_{ret} shrinks by $2n$ until it is empty.

They might define the traffic belonging to a slice using VLANs, MPLS, FlowVisor, or – as far as we are concerned – any set of headers. A common requirement is that traffic stay within its slice, and not leak to another slice. Leakage might cause a malfunction or lead to a security breach.

Header space analysis can (1) Help create new slices that are guaranteed to be isolated, and can (2) Detect when slices are leaking traffic. We consider each use case in turn.

Creating New Slices: Creating a new slice requires identification of a region of network space that does not overlap with regions belonging to existing slices. Consider an example of two slices, a and b with regions of network space $N_a, N_b \in \mathcal{N}$,

$$N_a = \{(\alpha_i, p_i)]_{p_i \in \mathcal{S}}\} \quad , \quad N_b = \{(\beta_i, p_i)]_{p_i \in \mathcal{S}}\}$$

where α are headers in N_a and β are headers in N_b, and $p_i \in \mathcal{S}$ are individual ports in each slice.

If the two slices do not overlap, they have no header space in common on any common port, i.e., $\alpha_i \cap \beta_i = \phi$, for all i. If they intersect, we can determine precisely where (which links) and how (which headers) by finding their intersection:

$$N_a \cap N_b = \{(\alpha_i \cap \beta_i, p_i)]_{p_i \in N_a \& p_i \in N_b}\}.$$

Set intersection could, for example, be used to statically verify that communication is allowed at one layer, or with one protocol but not with another. A simple check for overlap is extremely useful in any slicing environment (e.g. VLANs or FlowVisor) to check for run-time violations. The test can flag violations, or could be be used to create one slice to monitor another.

Detecting Leakage. Even if two slices do not overlap anywhere, packets can still leak from one slice to another when headers are rewritten. We can use a (more involved) algorithm to check whether packets can leak. If there is leakage, the algorithm finds the set of offending $(header, port)$ pairs.

Assume that slice a has reserved network space N_a = $\{(\alpha_i, p_i)]_{p_i \in \mathcal{S}}\}$, and the network transfer function of slice a is $\Psi_a(h, p)$. Slice a is only allowed to control packets belonging to its slice using Ψ_a. Leakage occurs when a packet in slice a at any switch-port can be rewritten to fall into the network space of another slice. If packets cannot leak at any switch-port, then they cannot leak anywhere. Therefore on each switch-port, we apply the network transfer function of slice a to its header space reservation, to generate all possible packet headers from slice a. Call this the *output header set*. If the output header space of slice a at any switch port, overlaps with any other slice, then there is the potential for leaks. Figure 5 graphically represents this check.

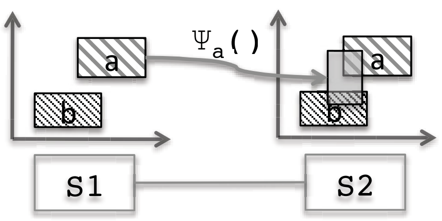


Figure 5: Detecting slice leakage. Although slice a and b have disjoint slice reservation on S_1 and S_2, but slice a's reservation on S_1 can leak to slice b's reservation os S_2 after it is rewritten by slice a's transfer function rules.

[1] shows that the complexity of both tests is $O(W^2N)$, where W is the maximum number of wildcard expressions used to describe any slice's reservation and N is the total number of slices in the network.

6 Implementation

We created a set of tools written in Python 2.6 - called Header Space Library or *Hassel* - that implement the techniques described above. The source code is available here [2]. Hassel's basic building block is a header space object that is a union of wildcard expressions which implements basic set operations. In Hassel, *transfer function objects* that implement network transfer functions are configured by a set of rules; when given a header space object and port, a transfer function generates a list of output header space objects and ports. Transfer functions can be built from standard rules (i.e. by matching on an input port and wildcard expression), or from custom rules supplied by the programmer. Hassel allows the computation of the inverse of a transfer function. We also wrote a Cisco IOS parser that parse router configurations and command outputs and generates a transfer function object that models the static behavior of the router. The resulting automation was essential in analyzing Stanford's network.

Figure 6 is a block diagram of Hassel. For Cisco routers, we first use Cisco IOS commands to show the MAC-address table, the ARP Table, the Spanning Tree, the IP forwarding table, and the router configuration. The result is passed to the parser which builds transfer function objects which are then used by applications such as Loop Detection.

Our implementation employs five key optimizations marked with superscript indices in Figure 6 that are keyed to the rows in Table 1. We briefly describe all optimizations, defering details to [1]. Table 1 reports the impact of disabling each optimization in turn when analyzing Stanford's backbone network. For example, loop detection for a single port with all optimizations enabled took 11 seconds: however, disabling IP compression increased running time by 19x and disabling lazy subtrac-

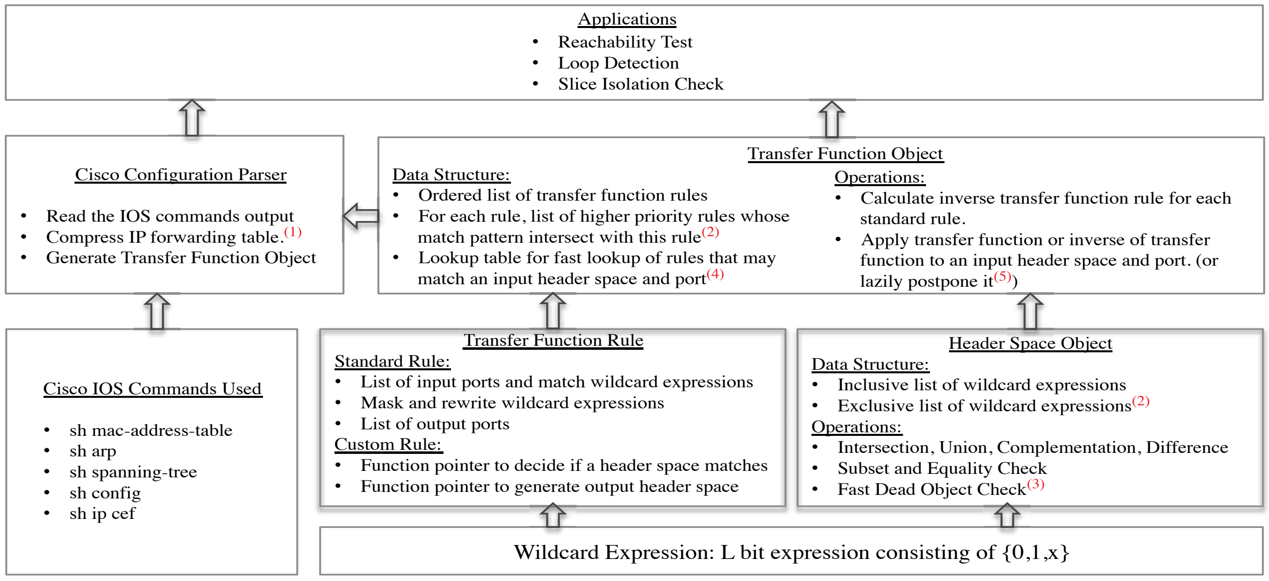

Applications
• Reachability Test
• Loop Detection
• Slice Isolation Check
Cisco Configuration Parser
• Read the IOS commands output
• Compress IP forwarding table.(1)
• Generate Transfer Function Object
Transfer Function Object
Data Structure:
• Ordered list of transfer function rules
• For each rule, list of higher priority rules whose match pattern intersect with this rule(2)
• Lookup table for fast lookup of rules that may match an input header space and port(4)
Operations:
• Calculate inverse transfer function rule for each standard rule.
• Apply transfer function or inverse of transfer function to an input header space and port. (or lazily postpone it(5))
Cisco IOS Commands Used
• sh mac-address-table
• sh arp
• sh spanning-tree
• sh config
• sh ip cef
Transfer Function Rule
Standard Rule:
• List of input ports and match wildcard expressions
• Mask and rewrite wildcard expressions
• List of output ports
Custom Rule:
• Function pointer to decide if a header space matches
• Function pointer to generate output header space
Header Space Object
Data Structure:
• Inclusive list of wildcard expressions
• Exclusive list of wildcard expressions(2)
Operations:
• Intersection, Union, Complementation, Difference
• Subset and Equality Check
• Fast Dead Object Check(3)
Wildcard Expression: L bit expression consisting of {0,1,x}

Figure 6: Header Space Library (Hassel) block diagram.

Disabled Optimization	T.F. Generation	Reach. Test	Loop Test
None	160s	12s	11s
(1) IP Table Compression	10.5x	15x	19x
(2) Lazy Subtraction	1x	>400x	>400x
(3) Dead Object Deletion	1x	8x	11x
(4) Lookup Based Search	0.9x	2x	2x
(5) Lazy T.F. evaluation	1x	1.2x	1.2x

Table 1: Impact of optimization techniques on the runtime of the reachability and loop detection algorithms.

tion inflated running time by 400x. Since the optimizations are orthogonal, the overall effect of all optimizations is around 10,000X, transforming Hassel from a toy to a tool.

IP Table Compression: We used IP forwarding table compression techniques in [7] to reduce the number of transfer function rules.

Lazy Subtraction: The simple geometric model, which assumes the rules in the transfer function are disjoint, can dramatically increase the number of rules. For example, if a router has two destination IP addresses: 10.1.1.x and 10.1.x.x and uses longest prefix match, our original definition requires the second entry to be converted to 8 disjoint rules. To avoid this, we extended the notion of a header space object to accept a union of wildcard expressions *minus a union of wildcard expressions*: $\cup\{w_i\} - \cup\{w_j\}$. Then, when we want to apply 10.1.x.x rule to the input header space, we simply *subtract* from the final result, the output generated by the first rule. Lazy subtraction allows delaying the expansion of terms during intermediate steps, only doing so at the end. As the table suggests, performance is dramatically improved.

Dead Object Deletion: At intermediate steps, header space objects often evaluate to empty, and should be removed. Lazy subtraction masks such empty objects; so we added a quick test to detect empty header space objects without explicit subtraction.

Lookup Based Search: To pass an input header space through a transfer function object, we must find which transfer function rules match the input header space. We avoid inefficient linear search via a lookup table that returns all wildcard rules that may intersect with the (possibly wildcarded) search key. In [1] we described the details of how we implemented such a table.

Lazy Evaluation of Transfer Function Rules: It is possible for the header space to grow as the cross-product of the rules. For example, if some boxes forward based on D destination address, while others filter based on S source IP address, the network transfer function can have $D \times S$ fragments. If two transfer functions are orthogonal, HSL uses commutativity of transfer functions to delay computation of one set of rules until the end.

Bookmarking Applied Transfer Function Rules: Both reachability and loop detection tests, require tracing backwards using the inverse transfer function. HSL "bookmarks" or memoizes the specific transfer rules applied to a header space object along the forward path. HSL saves time during the reverse path computation by only inverting the bookmarked rules.

7 Evaluation

In this section, we first demonstrate the functionality of *Hassel* on Stanford University's backbone network and report performance results of our reachability and loop detection algorithms on an enterprise network. Then we benchmark the performance of our slice isolation test on random slices created on Stanford backbone network which are similar to the existing VLAN slices. Finally, we showcase the applicability of our approach to new protocols. All of our tests are run on a Macbook Pro, with Intel core i7, 2.66Ghz quad core CPU and 4GB of RAM. Only two of the cores were in use during the tests.

7.1 Verification Of An Enterprise Network

We ran Hassel on Stanford University's backbone network. With a population of over 15,000 students, 2,000 faculty, and five /16 IPv4 subnets, Stanford is a relatively large enterprise network. Figure 7 shows the network that connects departments and student dorms to the outside word. There are 14 operational zone (OZ) routers at the bottom connected via 10 switches to 2 backbone routers which connect Stanford to the outside world. Overall, the network has more than 757,000 forwarding entries and 1,500 ACL rules. We do not provide exact IP addresses or ports to meet privacy concerns.

We had two experimental goals: we wished to demonstrate the utility of running Hassel checks, and we wished to measure Hassel's performance in a production network. When we generated box transfer functions, we chose not to include learned MAC address of end hosts. This allowed us to unearth problems that can be masked by learned MAC addresses but may surface when learned entries expire.

Checking for loops: We ran the loop detection test on the entire backbone network by injecting test packets from 30 ports. It took 151 seconds to compress the forwarding table and generate transfer functions, and 560 seconds to run loop detection test for all 30 ports. IP table compression reduced the forwarding entries to around 4,200.[7]

The loop detection test *found 12 infinite loop paths* (ignoring TTL), such as path L1 in Figure 7, for packets destined to 10 different IP addresses. These loops are caused by interaction between spanning tree protocols of two VLANs: a packet broadcast on VLAN 1 can reach the leaves of the spanning tree of VLAN 1, where IP forwarding on a leaf nodes forwards it to VLAN 2. Then, the packet is broadcasted on VLAN 2 and is forwarded at the leaf of VLAN 2 back to the original VLAN where the process can continue.

Although IP TTL will terminate this process, if the TTL is 32 and the normal path length is 3, this consumes 10 times the normal resources during looping periods. More importantly, it shows how protocol interactions can lead to subtle problems. Each VLAN has a separate spanning tree that prevents loops but VLANs are often defined manually. More generally, individual protocols often contain automated mechanisms that guarantee correctness for the protocol by itself, but the interaction between protocols is often done manually. Such manual configuration often leads to errors which Hassel can check for; route redistribution [12] provides another example of how manual connection of different routing protocols can lead to errors.

[7]There were 4 routers which together had 733,000 forwarding entries because no default BGP route was received. As a result they kept one entry for every Internet subnet they knew of, but all with the same output port. IP compression reduced their table size by 3 orders of magnitude.

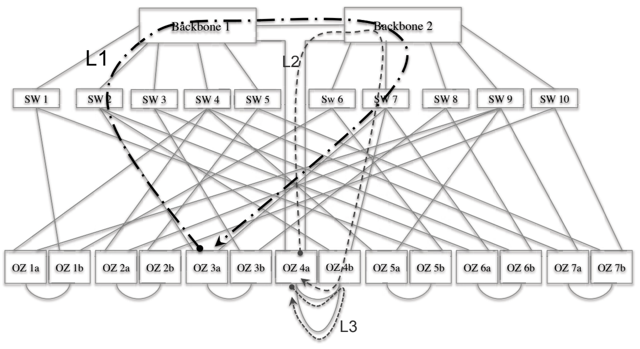


Figure 7: Topology of Stanford University's backbone network and 3 types of loops detected using Hassel. Overall, we found 26 loops on 14 loop paths. 10 of these loops, caused by packets destined to 10 IP addresses, are infinite loops masked by bridge learning. 16 other loops are single round loops.

We also found 4 other loop paths, similar to L1, L2 and L3 in Figure 7 for packets destined to 16 subnets. However, these loops were single-round loops, because when the packets return to the injection port, they are assigned to a VLAN not defined on that box, and hence will be dropped. Table 2 summarizes performance results. Note that we can trivially speed up the loop detection test by running each per-port test on a separate core.

Time to generate Network and Topology Transfer Function	151 s
Runtime of loop detection test (30 ports)	560 s
Average per port runtime	18.6 s
Max per port runtime	135 s
Min per port runtime	8 s
Average runtime of reachability test	13 s

Table 2: Runtime of loop detection and reachability tests on Stanford backbone network

Detecting possible configuration mistakes: As a second example, we considered a configuration mistake that could cause packets to loop between backbone router 1 and the Internet. Stanford owns the IP subnet 171.64.0.0/14. But not all of these IP addresses are currently in use. The backbone routers have an entry to route those IP addresses that are in use, to the correct OZ router. Also, the default route in the backbone routers (0.0.0.0/0) is to send packets to the internet. To avoid sending packets destined to the unused Stanford IP addresses to the outside world, the backbone routers have a manually installed null rule that drops all packets destined to 171.64.0.0/14, if they don't match any other rule.

Suppose that by mistake the null rule is set to drop 171.64.0.0/16 IP addresses (i.e., the /14 is fat-fingered to a /16). Assume that the ISP's router does not filter incoming traffic traffic from Stanford destined to Stanford. Then packets sent to unused addresses in 17.64.0.0/14 that are not in 171.64.0.0/16 will loop between the backbone routers and the ISP's router. We simulated this scenario, and the test successfully detected the loop in less than 10 minutes (as in Table 2). More importantly, the tool allowed the loop to be traced to the line in the configuration file that caused the error. In particular, the tool output shows that packets in the loop match the default 0.0.0.0/0 forwarding rule and not the 171.64.0.0/16 rule in the backbone router.

Verifying reachability to an OZ router: As a third example, we calculated the reachability function from the OZ router connected to the student dorms to the OZ router connected to the CS department. We verified that all the intended security restrictions, as commented by the admin in the config file were met. These restrictions included ports and IP addresses that were closed to outside users. Table 2 shows the run time for this test. We have heard from managers that many restrictions and ACLs were inserted by earlier managers and are still preserved because current managers are afraid to remove them. Hassel allows managers to do "What if" analysis to see the effect of deleting an ACL.

7.2 Checking Slice Isolation

Suppose we want to replace VLANs in the Stanford backbone network with the more flexible slices made possible by FlowVisor. Stanford's VLANs mostly carry traffic belonging to a particular subnet — e.g. VLAN 74 carries subnet 171.64.74.0/24. VLAN 74 is equivalent to a FlowVisor slice across the same routers with header space: $ip\_dst(h) = 171.64.74.0/24$ or $ip\_src(h) = 171.64.74.0/24$.

We would like to understand how quickly we can create new and flexible slices on-demand in the Stanford network. Recall from Section 5.3, we need to perform two checks:

1. When creating a slice, we need to make sure its header space does not overlap with an existing slice.
2. Whenever a rewrite action is added to a slice, we need to check that it cannot cause packet leakage.

We generated random slices with a topology similar to

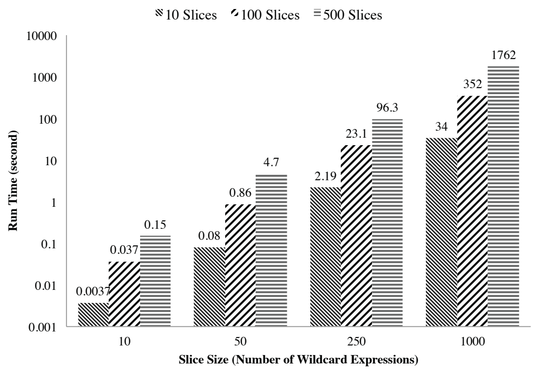


Figure 8: The time it takes to check if a new slice is isolated from other slices at reservation time.

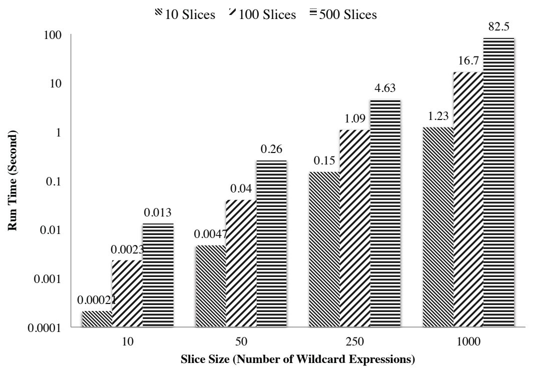


Figure 9: The time it takes to determine whether a new rewrite action will cause packets to leak between slices.

the existing VLANs, as follows: for each slice, we randomly pick two operational zones in Stanford together with all router ports and switches that connect them. Then we add random pieces of header space to each slice by picking X source or destination subnets of random prefix length (or random TCP ports) to union together. X, the number of wildcard expressions used to describe a slice, denotes the slice's *complexity*. While all existing VLAN slices in Stanford require fewer than 10 wildcard expressions[8], we explored the limits of performance by varying X from 10 to $1,000$. We ran experiments to create new slices of varying complexity, X, while there were 10, 100 or 500 existing slices.

In the first experiment, we create a new slice, and the checker verifies isolation by looking for intersection with all the existing slices. This test is done every time a new slice is created, and we hope it will complete in a few minutes or less. In the second experiment, we emulate the behavior of adding a new rewrite action. We generate random rewrite actions, and the checker checks to see if it could possibly cause a packet leak to another slice. This test has to be run every time a new rule is added, so it

[8]In most cases, two expressions suffice.

needs to be really fast.

Figure 8 and 9 shows the run time of our tests. As the figures suggest, if the slices are not very complex (can be explained with fewer than 50 wildcard expressions)[9], then the tests run almost instantly. Surprisingly, the tests are very fast even when there are 500 slices: more than adequate for existing networks. If there are only 10 or 100 slices, the checks can be done on very complex slices. The experimental run times matches the expected complexity in Section 5.3, which is *quadratic* in the number of wildcard expressions per slice and *linear* in the number of slices.

7.3 Debugging a Protocol Design

This section describes a plausible scenario in which a loop is caused by a protocol design mistake. The scenario allows the loop size to be parameterized to examine how detection time varies with loop size. It also showcases how HSL can model IP options and other variable length fields using custom transfer function rules.

In our scenario, Alice – a networking researcher – invents a new loose source routing protocol, IP*. IP* allows a source to specify the sequence of middle boxes that a packet must pass through. Alice's protocol has the header format shown in Figure 10.a. IP* works exactly like normal IP, except that it updates the header at the first router where a packet enters the IP* network. Figure 10.b shows an example of header update for a stack size of three. The header update operation sets the current source address to the "sender IP address" field, rewrites the destination IP address to the address at the top of the stack, and rotates all the IP addresses in the stack. After processing, a destination middlebox swaps the destination and source IP addresses and resends the packet to a router. Alice designs IP* to allow tunneling across existing IP networks.

Alice tries IP* in the network topology of Figure 11 and verifies that packets are successfully routed via middle boxes $M1$ and $M2$. Figure 11 also shows how the packet header changes as it passes through $M1$ and $M2$ to final destination DST in 6 steps. At DST, the stack contains the IP address of all middleboxes visited.

To continue her verification of IP*, Alice tries the more complex network in Figure 12 where the destination is attached to a different IP* network than the source. Instead of deploying a real network, she uses the loop detection algorithm from Section 5.2 and finds several loops. The most interesting one is an infinite loop consisting of the two strange loops below:

1) $R_2 \rightarrow R_5 \rightarrow M_2 \rightarrow R_5 \rightarrow R_2 \rightarrow R_4 \rightarrow M_1 \rightarrow R_4 \rightarrow R_2 \rightarrow R_3 \rightarrow R_6 \rightarrow R_3 \rightarrow R_2$

2) $R_2 \rightarrow R_4 \rightarrow M_1 \rightarrow R_4 \rightarrow R_2 \rightarrow R_5 \rightarrow M_2 \rightarrow R_5 \rightarrow R_2 \rightarrow R_3 \rightarrow R_6 \rightarrow R_3 \rightarrow R_2$

[9]This includes wildcard expressions that are included in, or excluded from, the definition of a slice.

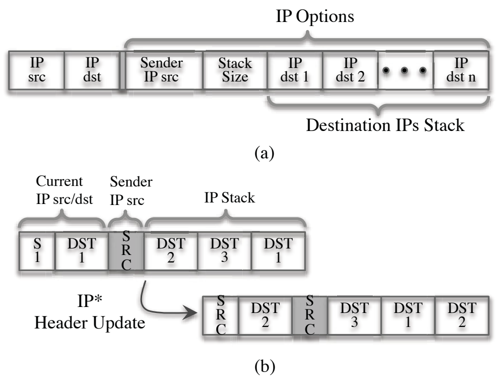


Figure 10: (a) Header format for IP* protocol. (b) IP* stack rotation: 1. The packet's IP source is replaced by the Sender IP source. 2. The packet's IP destination is replaced by the top of the stack. 3. The stack is rotated.

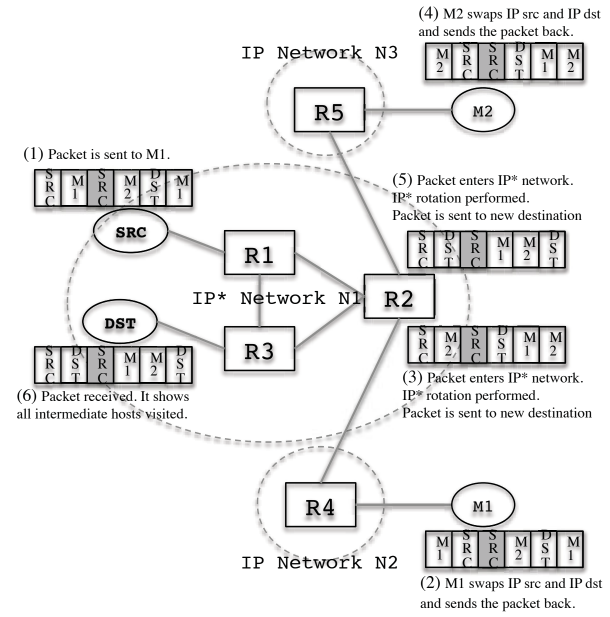

(4) M2 swaps IP src and IP dst and sends the packet back.
IP Network N3
R5
M2
(1) Packet is sent to M1.
(5) Packet enters IP* network. IP* rotation performed. Packet is sent to new destination
SRC
R1
IP* Network N1
R2
DST
R3
(6) Packet received. It shows all intermediate hosts visited.
(3) Packet enters IP* network. IP* rotation performed. Packet is sent to new destination
R4
M1
IP Network N2
(2) M1 swaps IP src and IP dst and sends the packet back.

Figure 11: Example of an IP* network where a packet sent from SRC to DST visits middleboxes $M1$ and $M2$. The figure labels 6 steps of packet processing along with the transformed header at each step. $R2$ is the entry point to the IP* network which performs IP* header updates.

Alice now realizes that having a second IP* network in the path can cause loops. She removes the concept of a "first" router, instead adding a pointer that describes the middlebox to visit next. We should be clear that by no means are we proposing IP* as a viable protocol. Instead, we hope this example suggests that Hassel could be a useful tool for protocol designers as well as network managers. While this particular loop could be caught by a simulation, if there were many sources and destinations and the loop was caused by a more obscure pre-

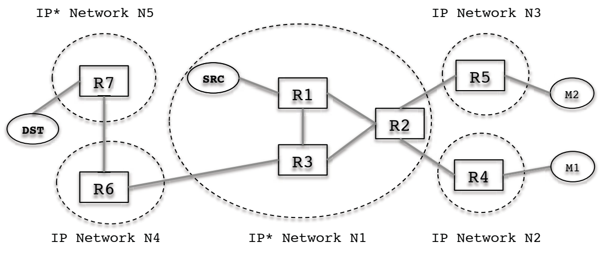


Figure 12: Alice's second network topology can cause infinite loops.

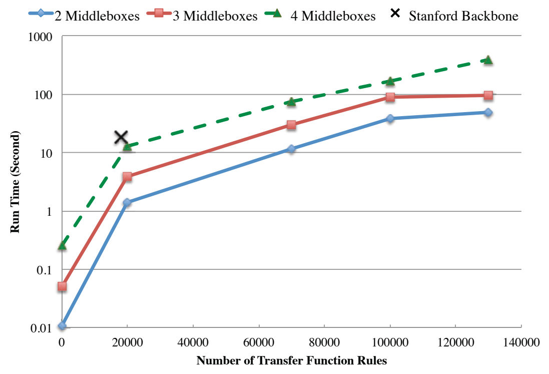


Figure 13: Running time of loop detection algorithm on Ip* network

```
def detect_loop(NTF, TTF, ports, test_packet):
    loops = []
    for port in ports:
        propagation = []
        p_node = {}
        p_node["hdr"] = test_packet
        p_node["port"] = port
        p_node["visits"] = []
        p_node["hs_history"] = []
        propagation.append(p_node)

        while len(propagation)>0:
            tmp_propag = []
            for p_node in propagation:
                next_hp = NTF.T(p_node["hdr"],p_node["port"])
                for (next_h,next_ps) in next_hp:
                    for next_p in next_ps:
                        linked = TTF.T(next_h,next_p)
                        for (linked_h,linked_ports) in linked:
                            for linked_p in linked_ports:
                                new_p_node = {}
                                new_p_node["hdr"] = linked_h
                                new_p_node["port"] = linked_p
                                new_p_node["visits"] = list(p_node["visits"])
                                new_p_node["visits"].append(p_node["port"])
                                new_p_node["hs_history"] = list(p_node["hs_history"])
                                new_p_node["hs_history"].append(p_node["hdr"])
                                if len(new_p_node["visits"]) > 0 \
                                    and new_p_node["visits"][0] == linked_p:
                                        loops.append(new_p_node)
                                        print "loop detected"
                                elif linked_p not in new_p_node["visits"]:
                                    tmp_propag.append(new_p_node)
            propagation = tmp_propag

    return loops
```

Figure 14: Loop Detection Code

condition, then a simulation may not uncover the loop. By contrast, static checking using Hassel will find all loops.

Figure 13 shows the *per-port* performance of our infinite loop detection algorithm for the ports participating in the loops. We varied the number of middle boxes connected to R_2 (e.g., $M1$ and $M2$) and the number of router forwarding entries. For four middle boxes, the loop has a length of 72 nodes! Finding a large and complex loop in less than four minutes — for a network with 100,000 forwarding rules and *custom* actions — using less than 50 lines[10] of python code (figure 14) demonstrates the power of the Header Space framework.

8 Limitations

Header space analysis is designed for static analysis, to detect forwarding and configuration errors. It is no panacea, serving as one tool among many needed by protocol designers, software developers, and network operators. For example, while header space analysis might tell us that a routing algorithm is broken because routing tables are inconsistent, it does not tell us why. Even if the routing tables are consistent, header space analysis offers no clues as to whether routing is efficient or meets the objectives of the designer. Despite this, header space analysis could play a similar role in networks as post-layout verification tools do in chip design, or static analysis checkers do in compilation. It checks the low level output against a set of universal invariants, without understanding the intent or aspirations of the protocol designer. To analyze protocol correctness, other approaches, such as [14] should be used.

Similarly, while our approach can pinpoint the specific entry in the forwarding table or line in the configuration file that causes a problem, it does not tell us how or why those entries were inserted or how they will be evolved as the box receives future messages. Finally, like all static checkers, our formalism and tools cannot deal well with churn in the network, except to periodically run it based on snapshots: thus it can only detect problems that persist longer than the sampling period.

9 Related Work

The notion of a transfer function in our work is similar to ASE mapping defined in axiomatic routing [14], where the authors develop tools to analyze a variety of protocols. Header space analysis makes no attempt to analyze protocols; instead, it tackles the problem of independently checking if their output creates conflicts. Roscoe's predicate routing [8] introduces the notion of pushing a test packet (as used in our reachability and loop detection algorithms) when designing routing mechanisms, rather than for static checking as we do. Xie's reachability analysis [4] uses test packets to determine reachability (not loop detection) for the special case of TCP/IP networks. The static analysis tools described in [11, 9, 10] are designed specifically for TCP/IP firewalls, and Feamster's work in [15] finds reachability failures in BGP routers.

Header space analysis is broader in two ways. First, it is a framework that can identify a range of network configuration problems. Second, the algorithms developed in this framework are independent of protocols. Finally, note that model checking, SAT solvers, and Theorem Provers are other commonly used frameworks for veri-

[10]Not counting the underlying Hassel implementation

fication [16]. However, when these frameworks detect a violation of a specification (e.g., reachability) they are limited to providing a *single counterexample* and not the *full set* of failed packet headers that header space analysis provides.

10 Conclusions

Our paper introduces *Header Space Analysis*: a general framework for reasoning about arbitrary protocols, and for finding common failures, or accidents. By parsing routing and configuration tables automatically, we show that header space analysis can be used in existing networks where protocol interactions are increasingly complex. As we saw in the Stanford backbone and noted by [12], while individual protocols use automated mechanisms to prevent internal problems, managing many protocols simultaneously is a manual and error-prone business. Header Space Analysis can also be used in emerging networks, where new protocols can be added dynamically. It can give network operators the confidence to adopt new protocols, or new slicing mechanisms — the framework can be used to create comprehensive checkers that can be used by network operators to pro-actively avoid (or retroactively investigate) accidents.

Our personal story is that we set out to create a geometric model to better understand slicing. Along the way, we discovered how simple it is to analyze networks using Network Transfer Functions (Ψ). We found that checking for a given violation was surprisingly easy when expressed using this high level abstraction, allowing elegant expression and simple implementation as our code snippets (see Figure 14) suggest.

We have work to do to improve the performance of our prototype Hassel implementation. A first round of optimization reduced running time by five orders of magnitude, making Hassel perform well for production networks with a few dozen routers, adequate for most enterprises and campuses. With simple fixes (e.g. exploiting 64-bit arithmetic, using compiled languages rather than Python, and harnessing the parallelism of multicore chips) we expect another 2-3 orders of magnitude performance gain. We expect running time of a complete loop test for a campus backbone to be reduced from about 1,000 seconds to less than 10 seconds. Even more optimizations are apparent, such as using a Karnaugh-Map to reduce the size of header space after each transformation. There is also scope for checking updates incrementally, by analyzing loops and reachability once, and then seeing how a new rule (or slice) changes the result.

We have other work ahead of us: we would like to create tools that create test packets to dynamically sample header space to detect faults in an operational network. We also hope to explore the notion of how secure, or how fault-tolerant, a network is by finding the "distance" between the current status of the network, and different failure conditions — analogous to Hamming distance.

Accidents will happen in the best regulated of networks; but the judicious use of checkers such as ours can reduce their probability.

11 Acknowledgement

We would like to thank our shepherd, Ranveer Chandra and the anonymous reviewers for their valuable comments. We also thank our network admin, Johan van Reijendam, for collecting network data for us. The first author was supported by a Stanford Graduate Fellowship (SGF) while doing this research.

References

[1] P. Kazemian, G. Varghese, N. McKeown, *Header Space Analysis*, Technical Report, `http://stanford.edu/~kazemian/hsa.pdf`

[2] Header Space Library (Hassel) `http:/stanford.edu/~kazemian/hassel.tar.gz`

[3] T. V. Lakshman and D. Stiliadis, *High-Speed Policy-based Packet Forwarding Using Efficient Multi-dimensional Range Matching*, In SIGCOMM. 1998.

[4] G. Xie, J. Zhan, D. Maltz, H. Zhang, A. Greenberg, G. Hjalmtysson, and J. Rexford, *On Static Reachability Analysis of IP Networks*, In INFOCOM. 2005.

[5] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, *OpenFlow: Enabling Innovation in Campus Networks*, In ACM SIGCOMM Computer Communication Review, Volume 38, Number 2, 2008.

[6] R. Sherwood, G. Gibb, K.K Yap, G. Appenzeller, M. Casado, N. McKeown, G. Parulkar, *Can the Production Network Be the Test-bed?*, In OSDI. 2010.

[7] R. Draves, C. King, S. Venkatachary, B. Zill, *C*onstructing optimal IP routing tables, In INFOCOM. 1999.

[8] T. Roscoe, S. Hand, R. Isaacs, R. Mortier, P. Jardetzky *Predicate Routing: Enabling Controlled Networking* In HotNets. 2002.

[9] Y. Bartal, A. J. Mayer, K. Nissim, and A. Wool. *Firmato:A novel firewall management toolkit*, In IEEE Symposium on Security and Privacy. 1999.

[10] A. Mayer, A. Wool, and E. Ziskind, *Fang: A firewall analysis engine*, In IEEE Symposium on Security and Privacy. 2000.

[11] L. Yuan, J. Mai, Z. Su, H. Chen, C-N. Chuah, and P. Mohapatra, *FIREMAN: A Toolkit for Firewall Modeling and Analysis*, In IEEE Symposium on Security and Privacy. 2006.

[12] F. Le, G. Xie, D. Pei, J. Wang, and H. Zhang, *Shedding Light on the Glue Logic of the Internet Routing Architecture*, In SIGCOMM. 2008.

[13] F. Le, G. Xie, and H. Zhang, *Understanding Route Redistribution*, In IEEE ICNP. 2007.

[14] M. Karsten, S. Keshav , S. Prasad , M. Beg *An Axiomatic Basis for Communication* In SIGCOMM. 2007.

[15] N. Feamster, H. Balakrishnan, *Detecting BGP configuration faults with static analysis*, In NSDI. 2005.

[16] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. B. Godfrey, S. T. King, *Debugging the data plane with anteater* In SIGCOMM. 2011.

[17] E. M. Clarke, O. Grumberg, D. A. Peled, *Model Checking*, MIT Press, 1999.

[18] S. Brown, Z. Vranesic, *Fundamentals of Digital Logic with Verilog Design*, McGraw-Hill, 2003.

[19] Global Environment for Network Innovations (GENI), `http://www.geni.org`

[20] The Health Insurance Portability and Accountability Act (HIPAA), `http://www.hhs.gov/ocr/privacy/`

A NICE Way to Test OpenFlow Applications

Marco Canini*, Daniele Venzano*, Peter Perešíni*, Dejan Kostić*, and Jennifer Rexford†

*EPFL †Princeton University

Abstract

The emergence of OpenFlow-capable switches enables exciting new network functionality, at the risk of programming errors that make communication less reliable. The centralized programming model, where a single controller program manages the network, seems to reduce the likelihood of bugs. However, the system is inherently distributed and asynchronous, with events happening at different switches and end hosts, and inevitable delays affecting communication with the controller. In this paper, we present efficient, systematic techniques for testing unmodified controller programs. Our NICE tool applies model checking to explore the state space of the entire system—the controller, the switches, and the hosts. Scalability is the main challenge, given the diversity of data packets, the large system state, and the many possible event orderings. To address this, we propose a novel way to augment model checking with symbolic execution of event handlers (to identify representative packets that exercise code paths on the controller). We also present a simplified OpenFlow switch model (to reduce the state space), and effective strategies for generating event interleavings likely to uncover bugs. Our prototype tests Python applications on the popular NOX platform. In testing three real applications—a MAC-learning switch, in-network server load balancing, and energy-efficient traffic engineering—we uncover eleven bugs.

1 Introduction

While lowering the barrier for introducing new functionality into the network, Software Defined Networking (SDN) also raises the risks of software faults (or *bugs*). Even today's networking software—written and extensively tested by equipment vendors, and constrained (at least somewhat) by the protocol standardization process—can have bugs that trigger Internet-wide outages [1, 2]. In contrast, programmable networks will offer a much wider range of functionality, through software created by a diverse collection of network operators and third-party developers. The ultimate success of SDN, and enabling technologies like OpenFlow [3], depends on having effective ways to test applications in pursuit of achieving high reliability. In this paper, we present NICE, a tool that efficiently uncovers bugs in OpenFlow programs, through a combination of model checking and symbolic execution. Building on our position paper [4] that argues for automating the testing of OpenFlow applications, we introduce several new contributions summarized in Section 1.3.

1.1 Bugs in OpenFlow Applications

An OpenFlow network consists of a distributed collection of switches managed by a program running on a logically-centralized controller, as illustrated in Figure 1. Each switch has a flow table that stores a list of rules for processing packets. Each rule consists of a pattern (matching on packet header fields) and actions (such as forwarding, dropping, flooding, or modifying the packets, or sending them to the controller). A pattern can require an "exact match" on all relevant header fields (*i.e.*, a *microflow* rule), or have "don't care" bits in some fields (*i.e.*, a *wildcard* rule). For each rule, the switch maintains traffic counters that measure the bytes and packets processed so far. When a packet arrives, a switch selects the highest-priority matching rule, updates the counters, and performs the specified action(s). If no rule matches, the switch sends the packet header to the controller and awaits a response on what actions to take. Switches also send event messages, such as a "join" upon joining the network, or "port change" when links go up or down.

The OpenFlow controller (un)installs rules in the switches, reads traffic statistics, and responds to events. For each event, the controller program defines a handler, which may install rules or issue requests for traffic statistics. Many OpenFlow applications[1] are written on the NOX controller platform [5], which offers an OpenFlow

[1] In this paper, we use the terms "OpenFlow application" and "controller program" interchangeably.

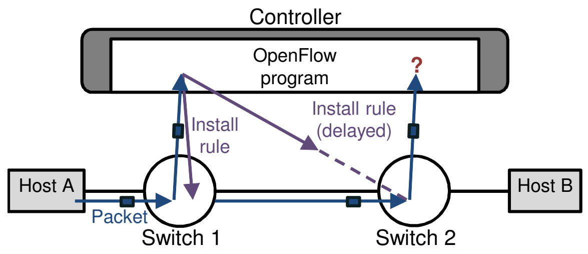


Figure 1: An example of OpenFlow network traversed by a packet. In a plausible scenario, due to delays between controller and switches, the packet does not encounter an installed rule in the second switch.

API for Python and C++ applications. These programs can perform arbitrary computation and maintain arbitrary state. A growing collection of controller applications support new network functionality [6–11], over OpenFlow switches available from several different vendors. Our goal is to create an efficient tool for systematically testing these applications. More precisely, we seek to discover violations of (network-wide) correctness properties due to bugs in the controller programs.

On the surface, the centralized programming model should reduce the likelihood of bugs. Yet, the system is inherently distributed and asynchronous, with events happening at multiple switches and inevitable delays affecting communication with the controller. To reduce overhead and delay, applications push as much packet-handling functionality to the switches as possible. A common programming idiom is to respond to a packet arrival by installing a rule for handling subsequent packets in the data plane. Yet, a race condition can arise if additional packets arrive while installing the rule. A program that implicitly expects to see just one packet may behave incorrectly when multiple arrive [4]. In addition, many applications install rules at multiple switches along a path. Since rules are not installed atomically, some switches may apply new rules before others install theirs. Figure 1 shows an example where a packet reaches an intermediate switch before the relevant rule is installed. This can lead to unexpected behavior, where an intermediate switch directs a packet to the controller. As a result, an OpenFlow application that works correctly most of the time can misbehave under certain event orderings.

1.2 Challenges of Testing OpenFlow Apps

Testing OpenFlow applications is challenging because the behavior of a program depends on the larger environment. The end-host applications sending and receiving traffic—and the switches handling packets, installing rules, and generating events—all affect the program running on the controller. The need to consider the larger environment leads to an extremely large state space, which "explodes" along three dimensions:

Large space of switch state: Switches run their own programs that maintain state, including the many packet-processing rules and associated counters and timers. Further, the set of packets that match a rule depends on the presence or absence of other rules, due to the "match the highest-priority rule" semantics. As such, testing OpenFlow applications requires an effective way to capture the large state space of the switch.

Large space of input packets: Applications are *data-plane* driven, *i.e.*, programs must react to a huge space of possible packets. The OpenFlow specification allows switches to match on source and destination MAC addresses, IP addresses, and TCP/UDP port numbers, as well as the switch input port; future generations of switches will match on even more fields. The controller can perform arbitrary processing based on other fields, such as TCP flags or sequence numbers. As such, testing OpenFlow applications requires effective techniques to deal with large space of inputs.

Large space of event orderings: Network events, such as packet arrivals and topology changes, can happen at any switch at any time. Due to communication delays, the controller may not receive events in order, and rules may not be installed in order across multiple switches. Serializing rule installation, while possible, would significantly reduce application performance. As such, testing OpenFlow applications requires efficient strategies to explore a large space of event orderings.

To simplify the problem, we could require programmers to use domain-specific languages that prevent certain classes of bugs. However, the adoption of new languages is difficult in practice. Not surprisingly, most OpenFlow applications are written in general-purpose languages, like Python, Java. Alternatively, developers could create abstract models of their applications, and use formal-methods techniques to prove properties about the system. However, these models are time-consuming to create and easily become out-of-sync with the real implementation. In addition, existing model-checking tools like SPIN [12] and Java PathFinder (JPF) [13] cannot be directly applied because they require explicit developer inputs to resolve the data-dependency issues and sophisticated modeling techniques to leverage domain-specific information. They also suffer state-space explosion, as we show in Section 7. Instead, we argue that testing tools should operate directly on *unmodified* OpenFlow applications, and leverage *domain-specific knowledge* to improve scalability.

1.3 NICE Research Contributions

To address these scalability challenges, we present NICE (*No bugs In Controller Execution*)—a tool that tests unmodified controller programs by automatically generating carefully-crafted streams of packets under many possible event interleavings. To use NICE, the programmer

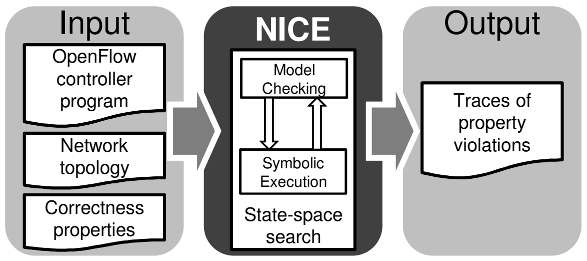


Figure 2: Given an OpenFlow program, a network topology, and correctness properties, NICE performs a state-space search and outputs traces of property violations.

supplies the controller program, and the specification of a topology with switches and hosts. The programmer can instruct NICE to check for generic correctness properties such as no forwarding loops or no black holes, and optionally write additional, application-specific correctness properties (*i.e.*, Python code snippets that make assertions about the global system state). By default, NICE systematically explores the space of possible system behaviors, and checks them against the desired correctness properties. The programmer can also configure the desired search strategy. In the end, NICE outputs property violations along with the traces to deterministically reproduce them. The programmer can also use NICE as a simulator to perform manually-driven, step-by-step system executions or random walks on system states.

Our design uses explicit state, software model checking [13–16] to explore the state space of the entire system—the controller program, the OpenFlow switches, and the end hosts—as discussed in Section 2. However, applying model checking "out of the box" does not scale. While simplified models of the switches and hosts help, the main challenge is the event handlers in the controller program. These handlers are data dependent, forcing model checking to explore all possible inputs (which doesn't scale) or a set of "important" inputs provided by the developer (which is undesirable). Instead, we extend model checking to *symbolically execute* [17, 18] the handlers, as discussed in Section 3. By symbolically executing the packet-arrival handler, NICE identifies equivalence classes of packets—ranges of header fields that determine unique paths through the code. NICE feeds the network a representative packet from each class by adding a state transition that injects the packet. To reduce the space of event orderings, we propose several domain-specific search strategies that generate event interleavings that are likely to uncover bugs in the controller program, as discussed in Section 4.

Bringing these ideas together, NICE combines model checking (to explore system execution paths), symbolic execution (to reduce the space of inputs), and search strategies (to reduce the space of event orderings). The programmer can specify correctness properties as snippets of Python code that operate on system state, or select from a library of common properties, as discussed in Section 5. Our NICE prototype tests unmodified applications written in Python for the popular NOX platform, as discussed in Section 6. Our performance evaluation in Section 7 shows that: (i) even on small examples, NICE is five times faster than approaches that apply state-of-the-art tools, (ii) our OpenFlow-specific search strategies reduce the state space by up to 20 times, and (iii) the simplified switch model brings a 7-fold reduction on its own. In Section 8, we apply NICE to three real OpenFlow applications and uncover *11* bugs. Most of the bugs we found are design flaws, which are inherently less numerous than simple implementation bugs. In addition, at least one of these applications was tested using unit tests. Section 9 discusses the trade-off between testing coverage and the overhead of symbolic execution. Section 10 discusses related work, and Section 11 concludes the paper with a discussion of future research directions.

2 Model Checking OpenFlow Applications

The execution of a controller program depends on the underlying switches and end hosts; the controller, in turn, affects the behavior of these components. As such, testing is not just a simple matter of exercising every path through the controller program—we must consider the state of the larger system. The needs to systematically explore the space of system states, and check correctness in each state, naturally lead us to consider *model checking* techniques. To apply model checking, we need to identify the system states and the transitions from one state to another. After a brief review of model checking, we present a strawman approach for applying model checking to OpenFlow applications, and proceed by describing changes that make it more tractable.

2.1 Background on Model Checking

Modeling the state space. A distributed system consists of multiple *components* that communicate asynchronously over message *channels*, *i.e.*, first-in, first-out buffers (*e.g.*, see Chapter 2 of [19]). Each component has a set of variables, and the *component state* is an assignment of values to these variables. The *system state* is the composition of the component states. To capture in-flight messages, the system state also includes the contents of the channels. A *transition* represents a change from one state to another (*e.g.*, due to sending a message). At any given state, each component maintains a set of enabled transitions, *i.e.*, the state's possible transitions. For each state, the enabled system transitions are the union of enabled transitions at all components. A *system execution* corresponds to a sequence of these transitions, and thus specifies a possible behavior of the system.

Model-checking process. Given a model of the state space, performing a search is conceptually straightforward. Figure 5 (non boxed-in text) shows the pseudo-code of the model-checking loop. First, the model checker initializes a stack of states with the initial state of the system. At each step, the checker chooses one state from the stack and one of its enabled transitions. After executing that transition, the checker tests the correctness properties on the newly reached state. If the new state violates a correctness property, the checker saves the error and the execution trace. Otherwise, the checker adds the new state to the set of explored states (unless the state was added earlier) and schedules the execution of all transitions enabled in this state (if any). The model checker can run until the stack of states is empty, or until detecting the first error.

2.2 Transition Model for OpenFlow Apps

Model checking relies on having a model of the system, *i.e.*, a description of the state space. This requires us to identify the states and transitions for each component—the controller program, the OpenFlow switches, and the end hosts. However, we argue that applying existing model-checking techniques imposes too much work on the developer and leads to an explosion in the state space.

2.2.1 Controller Program

Modeling the controller as a transition system seems relatively straightforward. A controller program is structured as a set of event handlers (*e.g.*, packet arrival and switch join/leave for the MAC-learning application in Figure 3), that interact with the switches using a standard interface, and these handlers execute atomically. As such, we can model the state of the program as the values of its global variables (*e.g.*, `ctrl_state` in Figure 3), and treat each event handler as a transition. To execute a transition, the model checker can simply invoke the associated event handler. For example, receiving a packet-in message from a switch enables the `packet_in` transition, and the model checker can execute the transition by invoking the corresponding event handler.

However, the behavior of event handlers is often data-dependent. In line 7 of Figure 3, for instance, the `packet_in` handler assigns `mactable` only for unicast source MAC addresses, and either installs a forwarding rule or floods a packet depending on whether or not the destination MAC address is known. This leads to different system executions. Unfortunately, model checking does not cope well with data-dependent applications (*e.g.*, see Chapter 1 of [19]). Since enumerating all possible inputs is intractable, a brute-force solution would require developers to specify a set of "relevant" inputs based on their knowledge of the application. Hence, a controller transition would be modeled as a pair consisting of an event handler and a concrete input. This is clearly undesirable. NICE overcomes this limitation

```
1  ctrl_state = {} # State of the controller is a global variable (a hashtable)
2  def packet_in(sw_id, inport, pkt, bufid): # Handles packet arrivals
3    mactable = ctrl_state[sw_id]
4    is_bcast_src = pkt.src[0] & 1
5    is_bcast_dst = pkt.dst[0] & 1
6    if not is_bcast_src:
7      mactable[pkt.src] = inport
8    if (not is_bcast_dst) and (mactable.has_key(pkt.dst)):
9      outport = mactable[pkt.dst]
10     if outport != inport:
11       match = {DL_SRC: pkt.src, DL_DST: pkt.dst, ↩
               DL_TYPE: pkt.type, IN_PORT: inport}
12       actions = [OUTPUT, outport]
13       install_rule(sw_id, match, actions, soft_timer=5, ↩
               hard_timer=PERMANENT) # 2 lines optionally
14       send_packet_out(sw_id, pkt, bufid) # combined in 1 API
15       return
16   flood_packet(sw_id, pkt, bufid)
17 def switch_join(sw_id, stats): # Handles when a switch joins
18   if not ctrl_state.has_key(sw_id):
19     ctrl_state[sw_id] = {}
20 def switch_leave(sw_id): # Handles when a switch leaves
21   if ctrl_state.has_key(sw_id):
22     del ctrl_state[sw_id]
```

Figure 3: Pseudo-code of a MAC-learning switch, based on the `pyswitch` application. The `packet_in` handler learns the input port associated with each non-broadcast source MAC address; if the destination MAC address is known, the handler installs a forwarding rule and instructs the switch to send the packet according to that rule; and otherwise floods the packet. The switch join/leave events initialize/delete a table mapping addresses to switch ports.

by using *symbolic execution* to automatically identify the relevant inputs, as discussed in Section 3.

2.2.2 OpenFlow Switches

To test the controller program, the system model must include the underlying switches. Yet, switches run complex software, and this is not the code we intend to test. A strawman approach for modeling the switch is to start with an existing reference OpenFlow switch implementation (*e.g.*, [20]), define the switch state as the values of all variables, and identify transitions as the portions of the code that process packets or exchange messages with the controller. However, the reference switch software has a large amount of state (*e.g.*, several hundred KB), not including the buffers containing packets and OpenFlow messages awaiting service; this aggravates the state-space explosion problem. Importantly, such a large program has many sources of nondeterminism and it is difficult to identify them automatically [16].

Instead, we create a switch model that omits inessential details. Indeed, creating models of some parts of the system is common to many standard approaches for applying model checking. Further, in our case, this is a one-time effort that does not add burden on the user. Following the OpenFlow specification [21], we view a switch as a set of communication channels, transitions that handle data packets and OpenFlow messages, and a flow table.

Simple communication channels: Each channel is a first-in, first-out buffer. Packet channels have an optionally-enabled fault model that can drop, duplicate, or reorder packets, or fail the link. The channel with the controller offers reliable, in-order delivery of OpenFlow messages, except for optional switch failures. We do not run the OpenFlow protocol over SSL on top of TCP/IP, allowing us to avoid intermediate protocol encoding/decoding and the substantial state in the network stack.

Two simple transitions: The switch model supports `process_pkt` and `process_of` transitions—for processing data packets and OpenFlow messages, respectively. We enable these transitions if at least one packet channel or the OpenFlow channel is non empty, respectively. A final simplification we make is in the `process_pkt` transition. Here, the switch dequeues the first packet from each packet channel, and processes *all* these packets according to the flow table. So, multiple packets at different channels are processed as a single transition. This optimization is safe because the model checker already systematically explores the possible orderings of packet arrivals at the switch.

Merging equivalent flow tables: A flow table can easily have two states that appear different but are semantically equivalent, leading to a larger search space than necessary. For example, consider a switch with two microflow rules. These rules do not overlap—no packet would ever match both rules. As such, the order of these two rules is not important. Yet, simply storing the rules as a list would cause the model checker to treat two different orderings of the rules as two distinct states. Instead, as often done in model checking, we construct a canonical representation of the flow table that derives a unique order of rules with overlapping patterns.

2.2.3 End Hosts

Modeling the end hosts is tricky, because hosts run arbitrary applications and protocols, have large state, and have behavior that depends on incoming packets. We could require the developer to provide the host programs, with a clear indication of the transitions between states. Instead, NICE provides simple programs that act as clients or servers for a variety of protocols including Ethernet, ARP, IP, and TCP. These models have explicit transitions and relatively little state. For instance, the default client has two basic transitions—`send` (initially enabled; can execute C times, where C is configurable) and `receive`—and a counter of sent packets. The default server has the `receive` and the `send_reply` transitions; the latter is enabled by the former. A more realistic refinement of this model is the mobile host that includes the `move` transition that moves the host to a new <switch, port> location. The programmer can also customize the models we provide, or create new models.

3 Symbolic Execution of Event Handlers

To systematically test the controller program, we must explore all of its possible transitions. Yet, the behavior of an event handler depends on the inputs (*e.g.*, the MAC addresses of packets in Figure 3). Rather than explore all possible inputs, NICE identifies which inputs would exercise different code paths through an event handler. Systematically exploring all code paths naturally leads us to consider *symbolic execution* (SE) techniques. After a brief review of symbolic execution, we describe how we apply symbolic execution to controller programs. Then, we explain how NICE combines model checking and symbolic execution to explore the state space effectively.

3.1 Background on Symbolic Execution

Symbolic execution runs a program with symbolic variables as inputs (*i.e.*, any values). The symbolic-execution engine tracks the use of symbolic variables and records the constraints on their possible values. For example, in line 4 of Figure 3, the engine learns that `is_bcast_src` is "`pkt.src[0] & 1`". At any branch, the engine queries a constraint solver for two assignments of symbolic inputs—one that satisfies the branch predicate and one that satisfies its negation (*i.e.*, takes the "else" branch)— and logically forks the execution to follow the feasible paths. For example, the engine determines that to reach line 7 of Figure 3, the source MAC address must have its eighth bit set to zero.

Unfortunately, symbolic execution does not scale well because the number of code paths can grow exponentially with the number of branches and the size of the inputs. Also, symbolic execution does not explicitly model the state space, which can cause repeated exploration of the same system state[2]. In addition, despite exploring all *code paths*, symbolic execution does not explore all *system execution paths*, such as different event interleavings. Techniques exist that can add artificial branching points to a program to inject faults or explore different event orderings [18, 22], but at the expense of extra complexity. As such, symbolic execution is not a sufficient solution for testing OpenFlow applications. Instead, NICE uses model checking to explore system execution paths (and detect repeated visits to the same state [23]), and symbolic execution to determine which inputs would exercise a particular state transition.

3.2 Symbolic Execution of OpenFlow Apps

Applying symbolic execution to the controller event handlers is relatively straightforward, with two exceptions.

[2]Unless expensive and possibly undecidable state-equivalence checks are performed.

First, to handle the diverse inputs to the `packet_in` handler, we construct *symbolic packets*. Second, to minimize the size of the state space, we choose a *concrete* (rather than symbolic) representation of controller state.

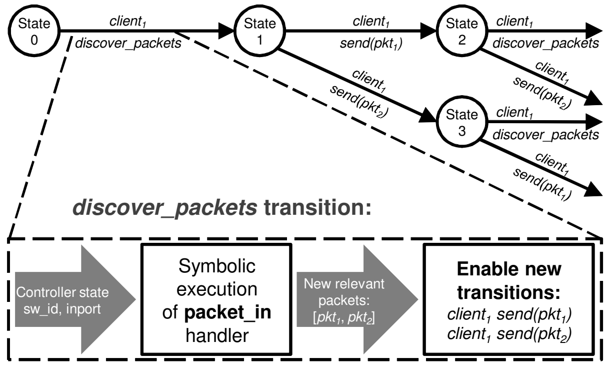
Symbolic packets. The main input to the `packet_in` handler is the incoming packet. To perform symbolic execution, NICE must identify which (ranges of) packet header fields determine the path through the handler. Rather than view a packet as a generic array of symbolic bytes, we introduce *symbolic packets* as our symbolic data type. A symbolic packet is a group of symbolic integer variables that each represents a header field. To reduce the overhead for the constraint solver, we maintain each header field as a lazily-initialized, individual symbolic variable (*e.g.*, a MAC address is a 6-byte variable), which reduces the number of variables. Yet, we still allow byte- and bit-level accesses to the fields. We also apply domain knowledge to further constrain the possible values of header fields (*e.g.*, the MAC and IP addresses used by the hosts and switches in the system model, as specified by the input topology).

Concrete controller state. The execution of the event handlers also depends on the controller state. For example, the code in Figure 3 reaches line 9 only for unicast destination MAC addresses stored in `mactable`. Starting with an empty `mactable`, symbolic execution cannot find an input packet that forces the execution of line 9; yet, with a non-empty table, certain packets could trigger line 9 to run, while others would not. As such, we must incorporate the global variables into the symbolic execution. We choose to represent the global variables in a concrete form. We apply symbolic execution by using these concrete variables as the initial state and by marking as symbolic the packets and statistics arguments to the handlers. The alternative of treating the controller state as symbolic would require a sophisticated type-sensitive analysis of complex data structures (*e.g.*, [23]), which is computationally expensive and difficult for an untyped language like Python.

3.3 Combining SE with Model Checking

With all of NICE's parts in place, we now describe how we combine model checking (to explore system execution paths) and symbolic execution (to reduce the space of inputs). At any given controller state, we want to identify the packets that each client should send—specifically, the set of packets that exercise all feasible code paths on the controller in that state. To do so, we create a special client transition called `discover_packets` that symbolically executes the `packet_in` handler. Figure 4 shows the unfolding of controller's state-space graph.

Symbolic execution of the handler starts from the initial state defined by (i) the concrete controller state



State 0
State 1
State 2
State 3
client1 discover_packets
client1 send(pkt1)
client1 send(pkt2)
client1 discover_packets
client1 send(pkt2)
client1 discover_packets
client1 send(pkt1)
discover_packets transition:
Controller state sw_id, inport
Symbolic execution of packet_in handler
New relevant packets: [pkt1, pkt2]
Enable new transitions: client1 send(pkt1) client1 send(pkt2)

Figure 4: Example of how NICE identifies relevant packets and uses them as new enabled send packet transitions of $client_1$. For clarity, the circled states refer to the controller state only.

(*e.g.*, State 0 in Figure 4) and (ii) a concrete "context" (*i.e.*, the switch and input port that identify the client's location). For every feasible code path in the handler, the symbolic-execution engine finds an equivalence class of packets that exercise it. For each equivalence class, we instantiate one *concrete packet* (referred to as the relevant packet) and enable a corresponding `send` transition for the client. While this example focuses on the `packet_in` handler, we apply similar techniques to deal with traffic statistics, by introducing a special `discover_stats` transition that symbolically executes the statistics handler with symbolic integers as arguments. Other handlers, related to topology changes, operate on concrete inputs (*e.g.*, the switch and port ids).

Figure 5 shows the pseudo-code of our search-space algorithm, which extends the basic model-checking loop in two main ways.

Initialization (lines 3-5)**:** For each client, the algorithm (i) creates an empty map for storing the relevant packets for a given controller state and (ii) enables the `discover_packets` transition.

Checking process (lines 12-18)**:** Upon reaching a new state, the algorithm checks for each client (line 15) whether a set of relevant packets already exists. If not, it enables the `discover_packets` transition. In addition, it checks (line 17) if the controller has a `process_stats` transition enabled in the newly-reached state, meaning that the controller is awaiting a response to a previous query for statistics. If so, the algorithm enables the `discover_stats` transition.

Invoking the `discover_packets` (lines 26-31) and `discover_stats` (lines 32-35) transitions allows the system to evolve to a state where new transitions become possible—one for each path in the packet-arrival or statistics handler. This allows the model checker to reach new controller states, allowing symbolic execution to again uncover new classes of inputs that enable additional transitions, and so on.

```
state_stack = []; explored_states = []; errors = []
initial_state = create_initial_state()
for client in initial_state.clients
    client.packets = {}
    client.enable_transition(discover_packets)
for t in initial_state.enabled_transitions:
    state_stack.push([initial_state, t])
while len(state_stack) > 0:
    state, transition = choose(state_stack)
    try:
        next_state = run(state, transition)
        ctrl = next_state.ctrl # Reference to controller in next_state
        ctrl_state = state(ctrl) # Stringified controller state in next_state
        for client in state.clients:
            if not client.packets.has_key(ctrl_state):
                client.enable_transition(discover_packets, ctrl)
        if process_stats in ctrl.enabled_transitions:
            ctrl.enable_transition(discover_stats, state, sw_id)
        check_properties(next_state)
        if next_state not in explored_states:
            explored_states.add(next_state)
            for t in next_state.enabled_transitions:
                state_stack.push([next_state, t])
    except PropertyViolation as e:
        errors.append([e, trace])
def discover_packets_transition(client, ctrl):
    sw_id, inport = switch_location_of(client)
    new_packets = SymbolicExecution(ctrl, packet_in, ↩
            context=[sw_id, inport])
    client.packets[state(ctrl)] = new_packets
    for packet in client.packets[state(ctrl)]:
        client.enable_transition(send, packet)
def discover_stats_transition(ctrl, state, sw_id):
    new_stats = SymbolicExecution(ctrl, process_stats, ↩
            context=[sw_id])
    for stats in new_stats:
        ctrl.enable_transition(process_stats, stats)
```

Figure 5: Pseudo-code of the state-space search algorithm used in NICE for finding errors. The highlighted parts, including the special "*discover*" transitions, are our additions to the basic model-checking loop.

By symbolically executing the controller event handlers, NICE can automatically infer the test inputs for enabling model checking without developer input, at the expense of some limitations in coverage of the system state space which we discuss later in Section 9.

4 OpenFlow-Specific Search Strategies

Even with our optimizations from the last two sections, the model checker cannot typically explore the entire state space, since this may be prohibitively large or even infinite. Thus, we propose domain-specific heuristics that substantially reduce the space of event orderings while focusing on scenarios that are likely to uncover bugs. Most of the strategies operate on the event interleavings produced by model checking, except for PKT-SEQ which reduces the state-space explosion due to the transitions uncovered by symbolic execution.

PKT-SEQ: Relevant packet sequences. The effect of discovering new relevant packets and using them as new enabled `send` transitions is that each end-host generates a potentially-unbounded tree of packet sequences. To make the state space finite and smaller, this heuristic reduces the search space by bounding the possible end host transitions (indirectly, bounding the tree) along two dimensions, each of which can be fine-tuned by the user. The first is merely *the maximum length of the sequence*, or in other words, the depth of the tree. Effectively, this also places a hard limit to the issue of infinite execution trees due to symbolic execution. The second is the *maximum number of outstanding packets*, or in other words, the length of a packet burst. For example, if $client_1$ in Figure 4 is allowed only a 1-packet burst, this heuristic would disallow both `send(`pkt_2`)` in State 2 and `send(`pkt_1`)` in State 3. Effectively, this limits the level of "packet concurrency" within the state space. To introduce this limit, we assign each end host with a counter c; when $c = 0$, the end host cannot send any more packet until the counter is replenished. As we are dealing with communicating end hosts, we adopt as default behavior to increase c by one unit for every received packet. However, this behavior can be modified in more complex end host models, *e.g.*, to mimic the TCP flow and congestion controls.

NO-DELAY: Instantaneous rule updates. When using this simple heuristic, NICE treats each communication between a switch and the controller as a single atomic action (*i.e.*, not interleaved with any other transitions). In other words, the global system runs in "lock step." This heuristic is useful during the early stages of development to find basic design errors, rather than race conditions or other concurrency-related problems. For instance, this heuristic would allow the developer to realize that installing a rule prevents the controller from seeing other packets that are important for program correctness. For example, a MAC-learning application that installs forwarding rules based only on the destination MAC address would prevent the controller from seeing some packets with new source MAC addresses.

UNUSUAL: Uncommon delays and reorderings. With this heuristic, NICE only explores event orderings with unusual and unexpected delays, with the goal of uncovering race conditions. For example, if an event handler in the controller installs rules in switches 1, 2, and 3, the heuristic explores transitions that reverse the order by allowing switch 3 to install its rule first, followed by switch 2 and then switch 1. This heuristic uncovers bugs like the example in Figure 1.

FLOW-IR: Flow independence reduction. Many OpenFlow applications treat different groups of packets independently; that is, the handling of one group is not affected by the presence or absence of another. In this case, NICE can reduce the search space by exploring only one relative ordering between the events affecting each group. To use this heuristic, the programmer provides `isSameFlow`, a Python function that takes two packets (and the switch and input port) as arguments and returns whether the packets belong to the same group. For example, in some scenarios different microflows are independent, whereas other programs may treat packets with different destination MAC addresses independently.

Summary. PKT-SEQ is complementary to other strategies in that it only reduces the number of `send` transitions rather than the possible kind of event orderings. PKT-SEQ is enabled by default and used in our experiments (unless otherwise noted). The other heuristics can be selectively enabled.

5 Specifying Application Correctness

Correctness is not an intrinsic property of a system—a specification of correctness states what the system should (or should not) do, whereas the implementation determines what it actually does. NICE allows programmers to specify correctness properties as Python code snippets, and provides a library of common properties (*e.g.*, no forwarding loops or blackholes).

5.1 Customizable Correctness Properties

Testing correctness involves asserting safety properties (*"something bad never happens"*) and liveness properties (*"eventually something good happens"*), defined more formally in Chapter 3 of [19]. Checking for safety properties is relatively easy, though sometimes writing an appropriate predicate over all state variables is tedious. As a simple example, a predicate could check that the collection of flow rules does not form a forwarding loop or a black hole. Checking for liveness properties is typically harder because of the need to consider a possibly infinite system execution. In NICE, we make the inputs finite (*e.g.*, a finite number of packets, each with a finite set of possible header values), allowing us to check some liveness properties. For example, NICE could check that, once two hosts exchange at least one packet in each direction, no further packets go to the controller (a property we call "StrictDirectPaths"). Checking this liveness property requires knowledge not only of the system state, but also which transitions have executed.

To check both safety and liveness properties, NICE allows correctness properties to (i) access the system state, (ii) register callbacks invoked by NICE to observe important transitions in system execution, and (iii) maintain local state. In our experience, these features offer enough expressiveness for specifying correctness properties. For ease of implementation, these properties are represented as snippets of Python code that make assertions about global system state. NICE invokes these snippets after each transition. For example, to check the StrictDirectPaths property, the code snippet would have local state variables that keep track of whether a pair of hosts has exchanged at least one packet in each direction, and would flag a violation if a subsequent packet triggers a `packet_in` event at the controller. When a correctness check signals a violation, the tool records the execution trace that recreates the problem.

5.2 Library of Correctness Properties

NICE provides a library of correctness properties applicable to a wide range of OpenFlow applications. A programmer can select properties from a list, as appropriate for the application. Writing these correctness modules can be challenging because the definitions must be robust to communication delays between the switches and the controller. Many of the definitions must intentionally wait until a "safe" time to test the property to prevent natural delays from erroneously triggering a violation of the property. Providing these modules as part of NICE can relieve the developers from the challenges of specifying correctness properties precisely, though creating any custom modules would require similar care.

• *NoForwardingLoops*: This property asserts that packets do not encounter forwarding loops, and is implemented by checking that each packet goes through any given <switch, input port> pair at most once.

• *NoBlackHoles*: This property states that no packets should be dropped in the network, and is implemented by checking that every packet that enters the network ultimately leaves the network or is consumed by the controller itself (for simplicity, we disable optional packet drops and duplication on the channels). To account for flooding, the property enforces a zero balance between the packet copies and packets consumed.

• *DirectPaths*: This property checks that, once a packet has successfully reached its destination, future packets of the same flow do not go to the controller. Effectively, this checks that the controller successfully establishes a direct path to the destination as part of handling the first packet of a flow. This property is useful for many OpenFlow applications, though it does not apply to the MAC-learning switch, which requires the controller to learn how to reach both hosts before it can construct unicast forwarding paths in either direction.

• *StrictDirectPaths*: This property checks that, after two hosts have successfully delivered at least one packet of a flow in each direction, no successive packets reach the controller. This checks that the controller has established a direct path in *both* directions between the two hosts.

• *NoForgottenPackets*: This property checks that all switch buffers are empty at the end of system execution. A program can easily violate this property by forgetting to tell the switch how to handle a packet. This can eventually consume all the available buffer space for packets awaiting controller instruction; after a timeout, the switch may discard these buffered packets. A short-running program may not run long enough for the queue of awaiting-controller-response packets to fill, but the *NoForgottenPackets* property easily detects these bugs.

6 Implementation Highlights

We have built a prototype implementation of NICE written in Python so as to seamlessly support OpenFlow controller programs for the popular NOX controller platform (which provides an API for Python).

As a result of using Python, we face the challenge of doing symbolic execution for a dynamic, untyped language. This task turned out to be quite challenging from an implementation perspective. To avoid modifying the Python interpreter, we implement a derivative technique of symbolic execution called *concolic execution* [24][3], which executes the code with concrete instead of symbolic inputs. Alike symbolic execution, it collects constraints along code paths and tries to explore all feasible paths. Another consequence of using Python is that we incur a significant performance overhead, which is the price for favoring usability. We plan to improve performance in a future release of the tool.

NICE consists of three parts: (i) a model checker, (ii) a concolic-execution engine, and (iii) a collection of models including the simplified switch and several end hosts. We now briefly highlight some of the implementation details of the first two parts: the model checker and concolic engine, which run as different processes.

Model checker details. To checkpoint and restore system state, NICE takes the approach of remembering the sequence of transitions that created the state and restores it by replaying such sequence, while leveraging the fact that the system components execute deterministically. State-matching is doing by comparing and storing hashes of the explored states. The main benefit of this approach is that it reduces memory consumption and, secondarily, it is simpler to implement. Trading computation for memory is a common approach for other model-checking tools (*e.g.*, [15, 16]). To create state hashes, NICE serializes the state via the cPickle module and applies the built-in hash function to the resulting string.

Concolic execution details. A key step in concolic execution is tracking the constraints on symbolic variables during code execution. To achieve this, we first implement a new "symbolic integer" data type that tracks assignments, changes and comparisons to its value while behaving like a normal integer from the program point of view. We also implement arrays (tuples in Python terminology) of these symbolic integers. Second, we reuse the Python modules that naturally serve for debugging and disassembling the byte-code to trace the program execution through the Python interpreter.

[3]Concolic stands for <u>conc</u>rete + symb<u>olic</u>.

Further, before running the code symbolically, we normalize and instrument it since, in Python, the execution can be traced at best with single code-line granularity. Specifically, we convert the source code into its abstract syntax tree (AST) representation and then manipulate this tree through several recursive passes that perform the following transformations: (i) we split composite branch predicates into nested if statements to work around shortcut evaluation, (ii) we move function calls before conditional expressions to ease the job for the STP constraint solver [25], (iii) we instrument branches to inform the concolic engine on which branch is taken, (iv) we substitute the built-in dictionary with a special stub that exposes the constraints, and (v) we intercept and remove sources of nondeterminism (*e.g.*, seeding the pseudo-random number generator). The AST tree is then converted back to source code for execution.

7 Performance Evaluation

Here we present an evaluation of how effectively NICE copes with the large state space in OpenFlow.

Experimental setup. We run the experiments on the simple topology of Figure 1, where the end hosts behave as follows: host A sends a "*layer-2 ping*" packet to host B which replies with a packet to A. The controller runs the MAC-learning switch program of Figure 3. We report the numbers of transitions and unique states, and the execution time as we increase the number of concurrent pings (a pair of packets). We run all our experiments on a machine set up with Linux 2.6.32 x86_64 that has 64 GB of RAM and a clock speed of 2.6 GHz. Our prototype implementation does not yet make use of multiple cores.

Benefits of simplified switch model. We first perform a full search of the state space using NICE as a depth-first search model checker (NICE-MC, without symbolic execution) and compare to NO-SWITCH-REDUCTION: doing model-checking without a canonical representation of the switch state. Effectively, this prevents the model checker from recognizing that it is exploring semantically equivalent states. These results, shown in Table 1, are obtained without using any of our search strategies. We compute ρ, a metric of state-space reduction due to using the simplified switch model, as $\frac{Unique(\text{NO-SWITCH-REDUCTION})-Unique(\text{NICE-MC})}{Unique(\text{NO-SWITCH-REDUCTION})}$.

We observe the following:

• In both samples, the number of transitions and of unique states grow roughly exponentially (as expected). However, using the simplified switch model, the unique states explored in NICE-MC only grow with a rate that is about half the one observed for NO-SWITCH-REDUCTION.

	NICE-MC			NO-SWITCH-REDUCTION			
Pings	**Transitions**	**Unique states**	**CPU time**	**Transitions**	**Unique states**	**CPU time**	ρ
2	470	268	0.94 [s]	760	474	1.93 [s]	0.38
3	12,801	5,257	47.27 [s]	43,992	20,469	208.63 [s]	0.71
4	391,091	131,515	36 [m]	2,589,478	979,105	318 [m]	0.84
5	14,052,853	4,161,335	30 [h]	-	-	-	-

Table 1: Dimensions of exhaustive search in NICE-MC vs. model-checking without a canonical representation of the switch state, which prevents recognizing equivalent states. Symbolic execution is turned off in both cases. NO-SWITCH-REDUCTION did not finish with five pings in four days.

• The efficiency in state-space reduction ρ scales with the problem size (number of pings), and is substantial (factor of seven for three pings).

Heuristic-based search strategies. Figure 6 illustrates the contribution of NO-DELAY and FLOW-IR in reducing the search space relative to the metrics reported for the full search (NICE-MC). We omit the results for UNUSUAL as they are similar. The state space reduction is again significant; about factor of four for three pings. In summary, our switch model and these heuristics result in a 28-fold state space reduction for three pings.

Comparison to other model checkers. Next, we contrast NICE-MC with two state-of-the-art model checkers, SPIN [12] and JPF [13]. We create system models in PROMELA and Java that replicate as closely as possible the system tested in NICE. Due to space limitations, we only briefly summarize the results and refer to [26] for the details:

• As expected, by using an abstract model of the system, SPIN performs a full search more efficiently than NICE. Of course, state-space explosion still occurs: *e.g.*, with 7 pings, SPIN runs of out memory. This validates our decision to maintain hashes of system states instead of keeping entire system states.

• SPIN's partial-order reduction (POR)[4], decreases the growth rate of explored transitions by only 18%. This is because even the finest granularity at which POR can be applied does not distinguish between independent flows.

• Taken "as is", JPF is already slower than NICE by a factor of 290 with 3 pings. The reason is that JPF uses Java threads to represent system concurrency. However, JPF leads to too many possible thread interleavings to explore even in our small example.

• Even with our extra effort in rewriting the Java model to explicitly expose possible transitions, JPF is 5.5 times slower than NICE using 4 pings.

These results suggest that NICE, in comparison to the other model-checkers, strikes a good balance between (i) capturing system concurrency at the right level of granularity, (ii) simplifying the state space and (iii) allowing testing of unmodified controller programs.

[4]POR is a well-known technique for avoiding exploring unnecessary orderings of transitions (*e.g.*, [27]).

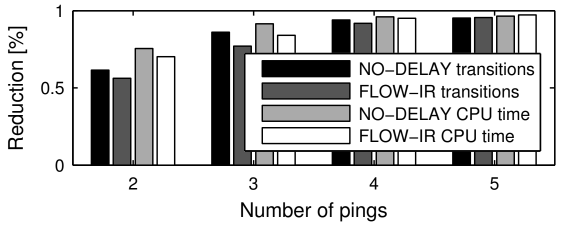


Figure 6: Relative state-space search reduction of our heuristic-based search strategies vs. NICE-MC.

8 Experiences with Real Applications

In this section, we report on our experiences applying NICE to three real applications—a MAC-learning switch, a server load-balancer, and energy-aware traffic engineering—and uncovering eleven bugs.

8.1 MAC-learning Switch (PySwitch)

Our first application is the `pyswitch` software included in the NOX distribution (98 LoC). The application implements MAC learning, coupled with flooding to unknown destinations, common in Ethernet switches. Realizing this functionality seems straightforward (*e.g.*, the pseudo-code in Figure 3), yet NICE automatically detects three violations of correctness properties.

BUG-I: Host unreachable after moving. This fairly subtle bug is triggered when a host B moves from one location to another. Before B moves, host A starts streaming to B, which causes the controller to install a forwarding rule. When B moves, the rule stays in the switch as long as A keeps sending traffic, because the soft timeout does not expire. As such, the packets do not reach B's new location. This serious correctness bug violates the *NoBlackHoles* property. If the rule had a *hard* timeout, the application would eventually flood packets and reach B at its new location; then, B would send return traffic that would trigger MAC learning, allowing future packets to follow a direct path to B. While this "bug fix" prevents persistent packet loss, the network still experiences *transient* loss until the hard timeout expires. Designing a new *NoBlackHoles* property that is robust to transient loss is part of our ongoing work.

BUG-II: Delayed direct path. The `pyswitch` also violates the *StrictDirectPaths* property, leading to suboptimal performance. The violation arises after a host A sends a packet to host B, and B sends a response packet to A. This is because `pyswitch` installs a forwarding rule in one direction—from the sender (B) to the destination (A), in line 13 of Figure 3. The controller does *not* install a forwarding rule for the other direction until seeing a subsequent packet from A to B. For a three-way packet exchange (*e.g.*, a TCP handshake), this performance bug directs 50% more traffic than necessary to the controller. Anecdotally, fixing this bug can easily introduce another one. The naïve fix is to add another `install_rule` call, with the addresses and ports reversed, after line 14, for forwarding packets from A to B. However, since the two rules are not installed atomically, installing the rules in this order can allow the packet from B to reach A before the switch installs the second rule. This can cause a subsequent packet from A to reach the controller unnecessarily. A correct fix would install the rule for traffic from A first, before allowing the packet from B to A to traverse the switch. With this "fix", the resulting program satisfies the *StrictDirectPaths* property.

BUG-III: Excess flooding. When we test `pyswitch` on a topology that contains a cycle, the program violates the *NoForwardingLoops* property. This is not surprising, since `pyswitch` does not construct a spanning tree.

8.2 Web Server Load Balancer

Data centers rely on load balancers to spread incoming requests over service replicas. Previous work created a load-balancer application that uses wildcard rules to divide traffic based on the client IP addresses to achieve a target load distribution [9]. The application can dynamically adjust the load distribution by installing new wildcard rules; during the transition, old transfers complete at their existing servers while new requests are handled according to the new distribution. We test this application with one client and two servers connected to a single switch. The client opens a TCP connection to a virtual IP address corresponding to the two replicas. In addition to the default correctness properties, we create an application-specific property *FlowAffinity* that verifies that all packets of a single TCP connection go to the same server replica. Here we report on the bugs NICE found in the original code (1209 LoC), which had already been unit tested to some extent.

BUG-IV: Next TCP packet always dropped after reconfiguration. Having observed a violation of the *NoForgottenPackets* property, we identified a bug where the application neglects to handle the "next" packet of each flow—for both ongoing transfers and new requests—after a change in the load-balancing policy. Despite correctly installing the forwarding rule for each flow, the application does *not* instruct the switch to forward the packet that triggered the `packet_in` handler. Since the TCP sender ultimately retransmits the lost packet, the program does successfully handle each Web request, making it hard to notice the bug. The bug degrades performance and, for a long execution trace, would ultimately exhaust the switch's space for buffering packets awaiting controller action.

BUG-V: Some TCP packets dropped after reconfiguration. After fixing **BUG-IV**, NICE detected another *NoForgottenPackets* violation due to a race condition. In switching from one load-balancing policy to another, the application sends multiple updates to the switch for each existing rule: (i) a command to remove the existing forwarding rule followed by (ii) commands to install one or more rules (one for each group of affected client IP addresses) that direct packets to the controller. Since these commands are not executed atomically, packets arriving between the first and second step do not match either rule. The OpenFlow specification prescribes that packets that do not match any rule should go to the controller. Although the packets go to the controller either way, these packets arrive with a different "reason code" (*i.e.*, `NO_MATCH`). As written, the `packet_in` handler ignores such (unexpected) packets, causing the switch to hold them until the buffer fills. This appears as packet loss to the end hosts. To fix this bug, the program should reverse the two steps, installing the new rules (perhaps at a lower priority) before deleting the existing ones.

BUG-VI: ARP packets forgotten during address resolution. Another *NoForgottenPackets* violation uncovered two bugs that are similar in spirit to the previous one. The controller program handles client ARP requests on behalf of the server replicas. Despite sending the correct reply, the program neglects to discard the ARP request packets from the switch buffer. A similar problem occurs for server-generated ARP messages.

BUG-VII: Duplicate SYN packets during transitions. A *FlowAffinity* violation detected a subtle bug that arises only when a connection experiences a duplicate (*e.g.*, retransmitted) SYN packet while the controller changes from one load-balancing policy to another. During the transition, the controller inspects the "next" packet of each flow, and assumes a SYN packet implies the flow is new and should follow the new load-balancing policy. Under duplicate SYN packets, some packets of a connection (arriving before the duplicate SYN) may go to one server, and the remaining packets to another, leading to a broken connection. The authors of [9] acknowledged this possibility (see footnote #2 in their paper), but only realized this was a problem after careful consideration.

8.3 Energy-Efficient Traffic Engineering

OpenFlow enables a network to reduce energy consumption [10,28] by selectively powering down links and redirecting traffic to alternate paths during periods of lighter load. REsPoNse [28] pre-computes several routing tables (the default is two), and makes an online selection for each flow. The NOX implementation (374 LoC) has an *always-on* routing table (that can carry all traffic under low demand) and an *on-demand* table (that serves additional traffic under higher demand). Under high load, the flows should probabilistically split evenly over the two classes of paths. The application learns the link utilizations by querying the switches for port statistics. Upon receiving a packet of a new flow, the `packet_in` handler chooses the routing table, looks up the list of switches in the path, and installs a rule at each hop.

For testing with NICE, we install a network topology with three switches in a triangle, one sender host at one switch and two receivers at another switch. The third switch lies on the on-demand path. We define the following application-specific correctness property:

• *UseCorrectRoutingTable*: This property checks that the controller program, upon receiving a packet from an ingress switch, issues the installation of rules to all and just the switches on the appropriate path for that packet, as determined by the network load. Enforcing this property is important, because if it is violated, the network might be configured to carry more traffic than it physically can, degrading the performance of end-host applications running on top of the network.

NICE found several bugs in this application:

BUG-VIII: The first packet of a new flow is dropped. A violation of *NoForgottenPackets* revealed a bug that is almost identical to **BUG-IV**. The `packet_in` handler installed a rule but neglected to instruct the switch to forward the packet that triggered the event.

BUG-IX: The first few packets of a new flow can be dropped. After fixing **BUG-VIII**, NICE detected another *NoForgottenPackets* violation at the second switch in the path. Since the `packet_in` handler installs an end-to-end path when the first packet of a flow enters the network, the program implicitly assumes that intermediate switches would never direct packets to the controller. However, with communication delays in installing the rules, the packet could reach the second switch before the rule is installed. Although these packets trigger `packet_in` events, the handler implicitly ignores them, causing the packets to buffer at the intermediate switch. This bug is hard to detect because the problem only arises under certain event orderings. Simply installing the rules in the reverse order, from the last switch to the first, is not sufficient—differences in the delays for installing the rules could still cause a packet to encounter a switch that has not (yet) installed the rule. A correct "fix" should either handle packets arriving at intermediate switches, or use "barriers" (where available) to ensure that rule installation completes at all intermediate hops before allowing the packet to depart the ingress switch.

BUG-X: Only on-demand routes used under high load. NICE detects a *CorrectRoutingTableUsed* violation that prevents on-demand routes from being used properly. The program updates an extra routing table in the port-statistic handler (when the network's perceived energy state changes) to either always-on or on-demand, in an effort to let the remainder of the code simply reference this extra table when deciding where to route a flow. Unfortunately, this made it impossible to split flows equally between always-on and on-demand routes, and the code directed all new flows over on-demand routes under high load. A "fix" was to abandon the extra table and choose the routing table on per-flow basis.

BUG	PKT-SEQ only	NO-DELAY	FLOW-IR	UNUSUAL
I	23 / 0.02	23 / 0.02	23 / 0.02	23 / 0.02
II	18 / 0.01	18 / 0.01	18 / 0.01	18 / 0.01
III	11 / 0.01	16 / 0.01	11 / 0.01	11 / 0.01
IV	386 / 3.41	1661 / 9.66	321 / 1.1	64 / 0.19
V	22 / 0.05	Missed	21 / 0.02	60 / 0.18
VI	48 / 0.05	48 / 0.06	31 / 0.04	49 / 0.07
VII	297k / 1h	191k / 39m	Missed	26.5k / 5m
VIII	23 / 0.03	22 / 0.02	23 / 0.03	23 / 0.02
IX	21 / 0.03	17 / 0.02	21 / 0.03	21 / 0.02
X	2893 / 35.2	Missed	2893 / 35.2	2367 / 25.6
XI	98 / 0.67	Missed	98 / 0.67	25 / 0.03

Table 2: Comparison of the number of transitions / running time to the first violation that uncovered each bug. Time is in seconds unless otherwise noted.

BUG-XI: Packets can be dropped when the load reduces. After fixing **BUG-IX**, NICE detected another violation of the *NoForgottenPackets*. When the load reduces, the program recomputes the list of switches in each always-on path. Under delays in installing rules, a switch not on these paths may send a packet to the controller, which ignores the packet because it fails to find this switch in any of those lists.

8.4 Overhead of Running NICE

In Table 2, we summarize how many seconds NICE took (and how many state transitions were explored) to discover the *first property violation* that uncovered each bug, under four different search strategies. Note the numbers are generally small because NICE quickly produces simple test cases that trigger the bugs. One exception, **BUG-VII**, is found in 1 hour by doing a PKT-SEQ-only search but UNUSUAL can detect it in just 5 minutes. Our search strategies are also generally faster than PKT-SEQ-only to trigger property violations, except in one case (**BUG-IV**). Also, note that there are no false positives in our case studies—every property violation is due to the manifestation of a bug—and only in few cases (**BUG-V**, **BUG-VII**, **BUG-X** and **BUG-XI**) the heuristic-based strategies experience false negatives. Expectedly, NO-DELAY, which does not consider rule installation delays, misses race condition bugs (27% missed bugs). **BUG-VII** is missed by FLOW-IR because the duplicate SYN is treated as a new independent flow (9% missed bugs).

Finally, the reader may find that some of the bugs we found—like persistently leaving some packets in the switch buffer—are relatively simple and their manifestations could be detected with run-time checks performed by the controller platform. However, the programmer would not know what caused them. For example, a run-time check that flags a "no forgotten packets" error due to **BUG-IV** or **BUG-V** would not tell the programmer what was special about this particular system execution that triggered the error. Subtle race conditions are very hard to diagnose, so having a (preferably small) example trace—like NICE produces—is crucial.

9 Coverage vs. Overhead Trade-Offs

Testing is inherently incomplete, walking a fine line between good coverage and low overhead. As part of our ongoing work, we want to explore further how to best leverage symbolic execution in NICE. We here discuss some limitations of our current approach.

Concrete execution on the switch: In identifying the equivalence classes of packets, the algorithm in Figure 5 implicitly assumes the packets reach the controller. However, depending on the rules already installed in the switch, some packets in a class may reach the controller while others do not. This leads to two limitations. First, if *no* packets in an equivalence class would go to the controller, generating a representative packet from this class was unnecessary. This leads to some loss in efficiency. Second, if *some* (but not all) packets go to the controller, we may miss an opportunity to test a code path through the handler by inadvertently generating a packet that stays in the "fast path" through the switches. This leads to some loss in both efficiency and coverage. We could overcome these limitations by extending symbolic execution to include our simplified switch model and performing "symbolic packet forwarding" across multiple switches. We chose not to pursue this approach because (i) symbolic execution of the flow-table code would lead to a path-explosion problem, (ii) including these variables would increase the overhead of the constraint solver, and (iii) rules that modify packet headers would further complicate the symbolic analysis. Still, we are exploring "symbolic forwarding" as future work.

Concrete global controller variables: In symbolically executing each event handler, NICE could miss complex dependencies *between* handler invocations. This is a byproduct of our decision to represent controller variables in a concrete form. In some cases, one call to a handler could update the variables in a way that affects the symbolic execution of a second call (to the same handler, or a different one). Symbolic execution of the second handler would start from the *concrete* global variables, and may miss an opportunity to recognize additional constraints on packet header fields. We could overcome this limitation by running symbolic execution across multiple handler invocations, at the expense of a significant explosion in the number of code paths. Or, we could revisit our decision to represent controller variables in a concrete form. As future work, we are considering ways to efficiently represent global variables symbolically.

Infinite execution trees in symbolic execution: Despite its many advantages, symbolic execution can lead to infinite execution trees [23]. In our context, an infinite state space arises if each state has at least one input that modifies the controller state. This is an inherent limitation of symbolic execution, whether applied independently or in conjunction with model checking. To address this limitation, we explicitly bound the state space by limiting the size of the input (*e.g.*, a limit on the number of packets) and devise OpenFlow-specific search strategies that explore the system state space efficiently. These heuristics offer a tremendous improvement in efficiency, at the expense of some loss in coverage.

10 Related Work

Bug finding. While model checking [12–16] and symbolic execution [17, 18, 24] are automatic techniques, a drawback is that they typically require a closed system, *i.e.*, a system (model) together with its environment. Typically, the creation of such environment is a manual process (*e.g.*, [22]). NICE re-uses the idea of model checking—systematic state-space exploration—and combines it with the idea of symbolic execution—exhaustive path coverage—to avoid pushing the burden of modeling the environment on the user. Also, NICE is the first to demonstrate the applicability of these techniques for testing the dynamic behavior of OpenFlow networks. Finally, NICE makes a contribution in managing state-space explosion for this specific domain.

Khurshid *et al.* [23] enable a model checker to perform symbolic execution. Both our and their work share the spirit of using symbolic variables to represent data from very large domains. Our approach differs in that it uses symbolic execution in a selective way for uncovering possible transitions given a certain controller state. As a result, we (i) reduce state-space explosion due to feasible code paths because not all code is symbolically executed, and (ii) enable matching of concrete system states to further reduce the search of the state space.

OpenFlow and network testing. Frenetic [29] is a domain-specific language for OpenFlow that aims to eradicate a large class of programming faults. Using Frenetic requires the network programmer to learn extensions to Python to support the higher-layer abstractions.

OFRewind [30] enables recording and replay of events for troubleshooting problems in production networks due to closed-source network devices. However, it does not automate the testing of OpenFlow controller programs.

Mai *et al.* [31] use static analysis of network devices forwarding information bases to uncover problems in the data plane. FlowChecker [32] applies symbolic model checking techniques on a manually-constructed network model based on binary decision diagrams to detect misconfigurations in OpenFlow forwarding tables. We view these works as orthogonal to ours since they both aim to analyze a snapshot of the data plane.

Bishop *et al.* [33] examine the problem of testing the specification of end host protocols. NICE tests the network itself, in a new domain of software defined networks. Kothari *et al.* [34] use symbolic execution and developer input to identify protocol manipulation attacks for network protocols. In contrast, NICE combines model checking with symbolic execution to identify relevant test inputs for injection into the model checker.

11 Conclusion

We built NICE, a tool for automating the testing of OpenFlow applications that combines model checking and concolic execution in a novel way to quickly explore the state space of unmodified controller programs written for the popular NOX platform. Further, we devised a number of new, domain-specific techniques for mitigating the state-space explosion that plagues approaches such as ours. We contrast NICE with an approach that applies off-the-shelf model checkers to the OpenFlow domain, and demonstrate that NICE is five times faster even on small examples. We applied NICE to implementations of three important applications, and found 11 bugs. A release of NICE is publicly available at `http://code.google.com/p/nice-of/`.

Acknowledgments.

We thank our shepherd Arvind Krishnamurthy and the anonymous reviewers who provided excellent feedback. We are grateful to Stefan Bucur, Olivier Crameri, Johannes Kinder, Viktor Kuncak, Sharad Malik, and Darko Marinov for useful discussions and comments on earlier drafts of this work. The research leading to these results has received funding from the European Research Council under the European Union's Seventh Framework Programme (FP7/2007-2013) / ERC grant agreement 259110.

References

[1] AfNOG Takes Byte Out of Internet. `http://goo.gl/HVJw5`.

[2] Reckless Driving on the Internet. `http://goo.gl/otilX`.

[3] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. OpenFlow: Enabling Innovation in Campus Networks. *SIGCOMM Comput. Commun. Rev.*, 38:69–74, March 2008.

[4] M. Canini, D. Kostić, J. Rexford, and D. Venzano. Automating the Testing of OpenFlow Applications. In *WRiPE*, 2011.

[5] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker. NOX: Towards an Operating System for Networks. *SIGCOMM Comput. Commun. Rev.*, 38:105–110, July 2008.

[6] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. Gude, N. McKeown, and S. Shenker. Rethinking Enterprise Network Control. *IEEE/ACM Transactions on Networking*, 17(4), August 2009.

[7] A. Nayak, A. Reimers, N. Feamster, and R. Clark. Resonance: Dynamic Access Control for Enterprise Networks. In *WREN*, 2009.

[8] N. Handigol et al. Plug-n-Serve: Load-Balancing Web Traffic using OpenFlow, August 2009. SIGCOMM Demo.

[9] R. Wang, D. Butnariu, and J. Rexford. OpenFlow-Based Server Load Balancing Gone Wild. In *Hot-ICE*, 2011.

[10] B. Heller, S. Seetharaman, P. Mahadevan, Y. Yiakoumis, P. Sharma, S. Banerjee, and N. McKeown. ElasticTree: Saving Energy in Data Center Networks. In *NSDI*, 2010.

[11] D. Erickson et al. A Demonstration of Virtual Machine Mobility in an OpenFlow Network, August 2008. SIGCOMM Demo.

[12] G. Holzmann. *The Spin Model Checker - Primer and Reference Manual*. Addison-Wesley, Reading Massachusetts, 2004.

[13] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model Checking Programs. *Automated Software Engineering*, 10(2):203–232, 2003.

[14] M. Musuvathi and D. R. Engler. Model Checking Large Network Protocol Implementations. In *NSDI*, 2004.

[15] C. E. Killian, J. W. Anderson, R. Jhala, and A. Vahdat. Life, Death, and the Critical Transition: Finding Liveness Bugs in Systems Code. In *NSDI*, 2007.

[16] J. Yang, T. Chen, M. Wu, Z. Xu, X. Liu, H. Lin, M. Yang, F. Long, L. Zhang, and L. Zhou. MODIST: Transparent Model Checking of Unmodified Distributed Systems. In *NSDI*, 2009.

[17] C. Cadar, D. Dunbar, and D. R. Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *OSDI*, 2008.

[18] S. Bucur, V. Ureche, C. Zamfir, and G. Candea. Parallel Symbolic Execution for Automated Real-World Software Testing. In *EuroSys*, 2011.

[19] C. Baier and J.-P. Katoen. *Principles of Model Checking*. The MIT Press, 2008.

[20] Open vSwitch: An Open Virtual Switch. `http://openvswitch.org`.

[21] OpenFlow Switch Specification. `http://www.openflow.org/documents/openflow-spec-v1.1.0.pdf`.

[22] R. Sasnauskas, O. Landsiedel, M. H. Alizai, C. Weise, S. Kowalewski, and K. Wehrle. KleeNet: Discovering Insidious Interaction Bugs in Wireless Sensor Networks Before Deployment. In *IPSN*, 2010.

[23] S. Khurshid, C. S. Păsăreanu, and W. Visser. Generalized Symbolic Execution for Model Checking and Testing. In *TACAS*, 2003.

[24] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed Automated Random Testing. In *PLDI*, 2005.

[25] V. Ganesh and D. L. Dill. A Decision Procedure for Bit-Vectors and Arrays. In *CAV*, 2007.

[26] P. Perešíni and M. Canini. Is Your OpenFlow Application Correct? In *CoNEXT Student Workshop*, 2011.

[27] C. Flanagan and P. Godefroid. Dynamic Partial-Order Reduction for Model Checking Software. In *POPL*, 2005.

[28] N. Vasić, D. Novaković, S. Shekhar, P. Bhurat, M. Canini, and D. Kostić. Identifying and using energy-critical paths. In *CoNEXT*, 2011.

[29] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker. Frenetic: A Network Programming Language. In *ICFP*, 2011.

[30] A. Wundsam, D. Levin, S. Seetharaman, and A. Feldmann. OFRewind: Enabling Record and Replay Troubleshooting for Networks. In *USENIX ATC*, 2011.

[31] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. B. Godfrey, and S. T. King. Debugging the Data Plane with Anteater. In *SIGCOMM*, 2011.

[32] E. Al-Shaer and S. Al-Haj. FlowChecker: Configuration Analysis and Verification of Federated OpenFlow Infrastructures. In *SafeConfig*, 2010.

[33] S. Bishop, M. Fairbairn, M. Norrish, P. Sewell, M. Smith, and K. Wansbrough. Rigorous Specification and Conformance Testing Techniques for Network Protocols, as applied to TCP, UDP, and Sockets. In *SIGCOMM*, 2005.

[34] N. Kothari, R. Mahajan, T. Millstein, R. Govindan, and M. Musuvathi. Finding Protocol Manipulation Attacks. In *SIGCOMM*, 2011.

Toward Predictable Performance in Software Packet-Processing Platforms

Mihai Dobrescu
EPFL, Switzerland

Katerina Argyraki
EPFL, Switzerland

Sylvia Ratnasamy
UC Berkeley

Abstract

To become a credible alternative to specialized hardware, general-purpose networking needs to offer not only flexibility, but also predictable performance. Recent projects have demonstrated that general-purpose multicore hardware is capable of high-performance packet processing, but under a crucial simplifying assumption of uniformity: all processing cores see the same type/amount of traffic and run identical code, while all packets incur the same type of conventional processing (e.g., IP forwarding). Instead, we present a general-purpose packet-processing system that combines ease of programmability with predictable performance, while running a diverse set of applications and serving multiple clients with different needs. Offering predictability in this context is considered a hard problem because software processes contend for shared hardware resources—caches, memory controllers, buses—in unpredictable ways. Still, we show that, in our system, (a) the way in which resource contention affects performance is predictable and (b) the overall performance depends little on how different processes are scheduled on different cores. To the best of our knowledge, our results constitute the first evidence that, when designing software network equipment, flexibility and predictability are not mutually exclusive goals.

1 Introduction

In recent years, both practitioners and researchers have argued for building evolvable networks, whose functionality changes with the needs of its users and is not tied to particular hardware vendors [4, 6, 17, 22]. An inexpensive way of building such networks is to run a network-programming framework like Click [21] on top of commodity general-purpose hardware [6, 17, 22]. Sekar et al. recently showed that, in such a network, operators can reduce network provisioning costs by up to a factor of 2.5 by dynamically consolidating middlebox functionality, i.e., assigning packet-processing tasks to the available general-purpose devices so as to minimize resource consumption [26].

To become a credible alternative to specialized hardware, general-purpose networking needs to offer not only flexibility but also predictable performance: network operators are unlikely to accept the risk that an unlucky configuration could cause unpredictable drop in network performance, potentially leading to customer dissatisfaction and violations of service-level agreements. Several projects have demonstrated that general-purpose multicore hardware can perform packet processing at line rates of 10Gbps or more [16–18,22,23]. However, in all cases, this was achieved under a crucial simplifying assumption of uniformity: all processing cores see the same type/amount of traffic and run identical code, while all packets receive the same kind of conventional packet processing (e.g., IP forwarding or some particular form of encryption). This setup allowed for low-level tuning of the entire system to one particular, simple, uniform workload (e.g., manually setting buffer and batch sizes).

Building a general-purpose system that offers predictable performance is considered a hard problem, especially when this system needs to support an evolvable set of applications that are potentially developed by various third parties. Such a system may perform as expected under certain conditions, but then a change in workload or a software upgrade could cause unpredictable, potentially significant, performance degradation.

One important reason for this lack of predictability is the complicated way in which software processes running on the same hardware affect each other: false sharing [8], unnecessarily shared data structures [9], and contention for shared hardware resources (caches, memory controllers, buses) [30]. The last factor, in particular, has been the subject of extensive research for more than two decades: researchers have been working on predicting the effects of resource contention since the appearance of simultaneous multithreaded processors [5,10,12,28,29,31,32], yet, to the best of our knowledge, none of the proposed models have found their way into practice. And packet-processing workloads create ample opportunity for resource contention, as they move packets between network card, memory controller and last-level cache, all of which are shared among multiple cores in modern platforms.

We set out to design and build a packet-processing system that combines ease of programmability with predictable performance, while supporting a diverse set of applications and serving multiple clients, each of which may require different types/combinations of packet processing. To offer ease of programmability, we rely on the Click network-programming framework [21]. We do not introduce any additional programming constraints, operating-system modifications, or low-level tuning for particular workloads.

We present a Click-based packet-processing system, built on top of a 12-core Intel Westmere platform, that supports a diverse set of realistic packet-processing applications (Section 2). Given this setup, we first investigate how resource contention affects packet processing: does it cause significant performance drop? how does the drop depend on specific properties of the involved applications? (Section 3) Then we look at how to predict these effects in a practical manner (Section 4). We also explore whether it makes sense to use “contention-aware scheduling” [34], a technique that reduces the effects of resource contention by not scheduling together processes that are likely to contend for hardware resources (Section 5).

Our main contribution is to show that it is feasible to build a software packet-processing system that achieves predictable performance in the face of resource contention. We also show that contention-aware scheduling may not be worth the effort in the context of packet processing. More specifically, using simple offline profiling of each application running alone, we are able to predict the contention-induced performance drop suffered by each of the applications sharing our system, with an error smaller than 3%. Moreover, in our system, the maximum overall performance improvement achieved by using contention-aware scheduling is 2%—and that only in one particular corner case. We provide intuition behind these results, and we quantitatively argue that they are not artifacts of the Intel architecture, rather they should hold on any modern multicore platform.

We consider our results to be good news for all the ongoing efforts in general-purpose networking: To the best of our knowledge, they constitute the first evidence that, when designing software network equipment, flexibility does not have to come at the cost of predictability.

2 System Setup

In this section, after introducing our hardware setup, we describe the packet-processing applications that we use to evaluate our work (§2.1) and the software configuration of our platform (§2.2).

As a basis for our system, we use a 12-core general-purpose server, illustrated in Figure 1, running SMP-Click [11] version 1.7, on Linux kernel 2.6.24.7. Our server is equipped with two Intel Xeon 5660 processors, each with 6×2.8GHz cores and an integrated memory controller. The 6 cores of each processor share a 12MB

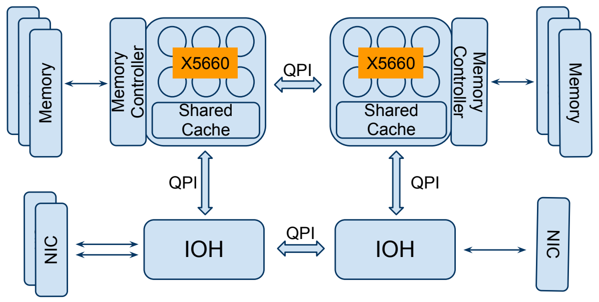


Figure 1: Overview of our platform’s architecture.

L3 cache, while each core has private L2 (256KB) and L1 (32KB for instructions and 32KB for data) caches. The two processors are interconnected via a 6.4GT/sec QuickPath interconnect (QPI). The server has 6 DDR3 memory modules (2GB each, 1333MHz) and 3 dual-port 10Gbps network interface cards (NICs) that use the Intel 82599 Niantic [2] chipset, resulting in 6×10Gbps ports.

2.1 Workloads

We designed our workloads to involve realistic forms of packet processing but make the job of predicting their performance as hard as possible. We implemented 5 forms of packet processing that are deployed in current network devices and cover a wide range of memory and CPU behavior. In our experiments, we craft the traffic that is processed by the system so as to maximize resource contention. More specifically, we implemented the following types of packet processing:

▷ *IP forwarding (IP).* Each packet is subjected to full IP forwarding, including longest-prefix-match lookup, checksum computation, and time-to-live (TTL) update. We use the RadixTrie lookup algorithm provided with the Click distribution and a routing-table of 128 000 entries. As input, we generate packets with random destination addresses, because this maximizes IP’s sensitivity to contention.

▷ *Monitoring (MON).* In addition to full IP forwarding, each packet is further subjected to NetFlow [1], a monitoring application. NetFlow collects statistics as follows: it applies a hash function to the IP and transport-layer header of each packet, uses the outcome to index a hash table with per-TCP/UDP-flow entries, and updates a few fields (a packet count and a timestamp) of the corresponding entry. As input, we generate packets with random IP addresses, such that the NetFlow hash table contains 100 000 entries. MON is a representative form of memory-intensive packet processing that benefits significantly from the L3 cache (both the routing table and the NetFlow hash table are cacheable data structures).

Flow	cycles per instruction	L3 references per sec (millions)	L3 hits per sec (millions)	cycles per packet	L3 references per packet	L3 misses per packet	L2 hits per packet
IP	1.33	25.85	20.21	1813	14.64	3.19	18.58
MON	1.43	27.26	21.32	2278	19.40	4.23	19.58
FW	1.63	2.71	2.13	23 907	20.22	4.29	56.10
RE	1.18	18.18	5.52	27 433	155.87	108.51	45.63
VPN	0.56	9.45	7.08	8679	25.63	6.41	30.71

Table 1: Characteristics of each type of packet processing during a solo run. Each number represents an average over 5 independent runs of the same experiment (the variance is negligible).

Also, it captures the nature of a wide range of packet-processing applications (applying a hash function to a portion of each packet and using the outcome to index and update a data structure).

▷ *Small firewall (FW).* In addition to full IP forwarding and NetFlow, each packet is further subjected to filtering: each packet is sequentially checked against 1000 rules and, if it matches any, it is discarded. We use sequential search (as opposed to a more sophisticated algorithm) because we consider a relatively small number of rules that can fit in the L2 cache. As input, we generate packets with random IP addresses that never match any of the rules; as a result, each packet is checked against all the rules, which maximizes FW's sensitivity to contention. This is a representative form of packet processing that benefits significantly from all the levels of the cache hierarchy.

▷ *Redundancy elimination (RE).* In addition to full IP forwarding and NetFlow, each packet is further subjected to RE [27], an application that eliminates redundant traffic. RE maintains a "packet store" (a cache of recently observed content) and a "fingerprint table" (that maps content fingerprints to packet-store entries). When a new packet is received, RE first updates the packet store, then uses the fingerprint table to check whether the packet includes a significant fraction of content cached in the packet store; if yes, instead of transmitting the packet as is, RE transmits an encoded version that eliminates this (recently observed) content. The assumption is that the device located at the other end of the link maintains a similar packet store and is able to recover the original contents of the packet. We implemented a packet store that can hold 1 second's worth of traffic and a fingerprint table with more than 4 million entries. This is a representative form of memory-intensive packet processing that does not significantly benefit from caching.

▷ *Virtual private network (VPN).* Each packet is subjected to full IP forwarding, NetFlow and AES-128 encryption. This is a representative form of CPU-intensive packet processing.

▷ *Synthetic processing (SYN).* For each received packet, we perform a configurable number of CPU operations (counter increments) and read a configurable number of random memory locations from a data structure that has the size of the L3 cache. We use this for profiling. We denote by *SYN_MAX* the most aggressive synthetic application that we were able to run on our system, which performs no other processing but consecutive memory accesses at the highest possible rate.

Table 1 summarizes the characteristics of each of these types of packet processing during a "solo" run (one core runs the packet-processing type, while all the other cores are idle). We use Oprofile [3] to count instructions, L2 hits, and L3 references and misses (we compute L3 hits as the difference between references and misses).

2.2 Software Configuration

Packet-processing parallelization. An important question is how packet processing should be parallelized among multiple cores. One possibility is the "pipeline" approach, where each packet is handled by multiple cores: one core reads it from memory, then passes it to another core for the first processing step, which passes it to another core for the second processing step, and so on. Another possibility is the "parallel" approach, where each packet is handled by a single core that reads the packet from memory and performs all the processing steps. The most recent general-purpose networking projects use the parallel approach, because it yields higher performance [16, 17]. However, a common criticism is that this is the case only for the simple, uniform workloads considered by these projects.

At first glance, choosing between the two approaches involves a trade-off (that we describe in detail in [14]). On the one hand, the parallel approach avoids passing the packet between different cores, hence eliminating synchronization and introducing fewer compulsory cache misses per packet. On the other hand, it requires that each core perform all the processing steps for each packet (hence accessing many different data structures), which may introduce a higher number of avoidable cache misses per packet due to cache contention. Hence, it seems intuitive that each approach would be best suited for different packet-processing applications.

After extensive experiments, we concluded that, in practice, there is no real trade-off between the two approaches: the parallel one is always better. This is because pipelining introduces several kinds of overhead that end up outweighing its potential benefit. For instance, passing socket-buffer descriptors, packet headers, and, potentially, payload between different cores results in compulsory cache misses. A less obvious source of overhead is memory management: Each core that handles packet reception uses a pre-allocated memory pool for storing packets. In a pipelined configuration, a packet is received by one core and transmitted by another; the transmitting core must recycle the buffer into the receiving core's pool of free buffers, and this requires extra synchronization between the two cores when removing/placing buffers in the pool. In our system, pipelining results in 10–15 extra cache misses per packet.

It *is* possible to craft a synthetic workload that performs better under the pipeline approach: it has to be a workload with enough processing steps and the right size of cacheable data structures such that running it on a parallel configuration results in more than 15 extra avoidable cache misses per packet than running it on a pipelined one. We describe such a workload in [14]: each received packet triggers more than 200 random memory accesses to a data structure that is *exactly* double the size of an L3 cache; even a small deviation from these numbers causes the advantage of the pipeline over the parallel approach to disappear. However, none of the realistic workloads that we looked at (including applications that involve deep packet inspection or redundancy elimination [27]) comes even close to such behavior.

Our configuration. We adopt the parallel approach for our system. Traffic arriving at each of our N network ports is split into Q receive queues. We refer to all traffic arriving at one receive queue as a *flow*; this is traffic that corresponds to one set of clients of our networking platform, all of which require the same type of packet processing. Each flow is handled by one core, which is responsible for reading the flow's packets from their receive queue, performing all the necessary processing, and writing them to the right transmit queue. Each core reads from its own receive queue(s) and writes to its own transmit queue(s), which are not shared with other cores. So, we have $N \cdot Q$ flows, each one assigned to one core, and each flow potentially involving a different type of packet processing.

In this paper, we focus on the scenario where each core processes one packet-processing flow: we use $N = 6$ ports and $Q = 2$ receive queues per port, so we have 12 different flows, each assigned to a separate core (we discuss this choice in Section 6). However, the Niantic cards support up to $Q = 128$ receive queues, so our prototype can, in principle, support hundreds of different flows.

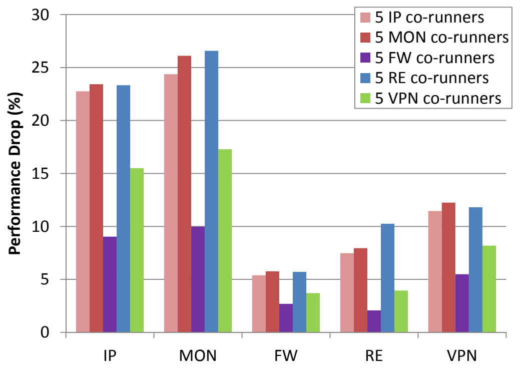
NUMA memory allocation. We ensure that each flow accesses its data "locally," i.e., through the memory controller that is directly connected to the processor handling the flow. We do this for two reasons: First, it has been shown (and we also verified experimentally) that accessing data remotely has a significant impact on memory-access latency [7], which, in our context, results in significant performance degradation. Second, to access data remotely, a flow has to use the processor interconnect, which can become a significant contention factor [7,34].

Theoretically, there are two scenarios where we might not be able to ensure local memory access, and we explain next why these do not arise in our system:

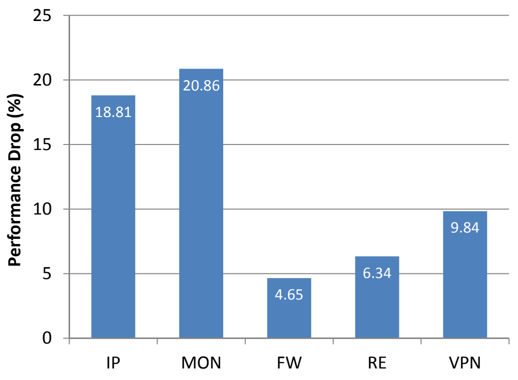
(a) Consider two flows, f_1 and f_2, that run on different processors and access the same data structure; one of the two flows will have to access the data remotely. This scenario does not arise in our system: If there are multiple flows that need to access the same data structure, we run all of them on the same processor. If there are more flows than per-processor cores that need to access the same data structure, we replicate the data structure across memory domains. We acknowledge that, in principle, such replication may break the semantics of a packet-processing application, however, we have not yet encountered any such (realistic) case. The closest we came was the redundancy-elimination application (described in §2.1), but, even there, it turned out that all the relevant data structures could be replicated across memory domains.

(b) Consider a flow f running on a core of processor P_1, accessing its data locally; due to contention-aware scheduling [7,34], we decide to move this flow to a core of processor P_2; as a result, f must now access its data remotely. This scenario does not arise in our system either: we will argue that, in our context, it does not make sense to perform contention-aware scheduling—one of the resulting benefits is that we do not have to deal with remote memory accesses.

Avoidable contention in the software stack. Before setting out to study resource contention between applications running on different cores, we sought to eliminate any form of "underlying" resource contention from our system, i.e., contention introduced not by the applications themselves but by the design of the underlying software stack: NIC driver, operating system, Click. We identified (a) false sharing and (b) unnecessary data sharing among multiple cores (e.g., the book-keeping data structures in the Niantic driver and the random seed of the Click random number generator were shared among multiple cores) as sources of such contention. We eliminated the former by padding data structures appropriately and the latter by replicating per-core data structures. Similar problems and fixes were recently presented in a scalability analysis of the Linux kernel [9].



(a) Performance drop suffered by each flow type in each scenario. For example, when a MON flow co-runs with 5 RE competitors, it suffers a drop of 27%.



(b) Average performance drop suffered by each flow type across all 5 scenarios that involve a target flow of that type. For example, the average performance drop suffered by the MON flow across all 5 scenarios (that involve a target MON flow) is 20.86%.

Figure 2: The effect of resource contention. For each pair of realistic flow types X and Y, we run an experiment in which a flow of type X co-runs with 5 flows of type Y. We measure the performance drop suffered by the flow of type X.

3 Understanding Contention

In this section, we identify which resources packet-processing flows mostly contend for (Section 3.1) and which flow properties determine the level of contention (Section 3.2), and we provide intuition behind our observations (Section 3.3).

We observe that the contention-induced performance drop suffered by a packet-processing flow is mostly determined by the number of last-level cache references per second performed by other flows sharing the same cache—not so much by the particular types of packet processing performed by these flows. As we will see, this observation is what enables us to do simple yet accurate performance prediction (Section 4).

In terms of terminology and notation, when we use the term "cache," we refer to the last-level cache shared by all cores of the same processor unless otherwise specified. We use "cache refs/sec" as an abbreviation for "cache references per second." We say that a flow *co-runs* with other flows when they all run on different cores of the same processor; we refer to all these flows as *co-runners*. In each experiment, we typically co-run 6 flows and study the performance drop suffered by one of these flows due to contention; we denote this flow by T (for "target") and each of its co-runners by C (for "competitor"). With respect to a flow T, we use the term *competing references* to refer to all the last-level cache references performed by this flow's co-runners.

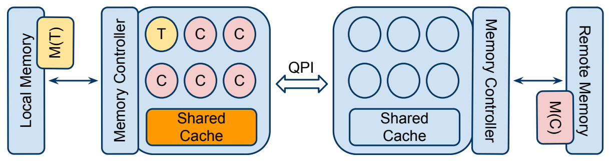
In all our experiments, we compute the performance drop suffered by a flow T due to contention with a set of competing flows as follows: First, we measure the throughput τ_s achieved by flow T during a solo run. Then we measure the throughput τ_c achieved by flow T when it co-runs with the set of competing flows. The contention-induced performance drop suffered by flow T is $\frac{\tau_s - \tau_c}{\tau_s}$. Each data point in our graphs represents the average over 5 independent runs of the same experiment (the variance is negligible).

We start by measuring the contention-induced performance drop suffered by realistic flow types in different scenarios (Figure 2). MON is the most sensitive type, suffering a performance drop of up to 27% (highest bar in Figure 2(a)), while FW suffers less than 6% in all experiments. RE is the most aggressive flow type, causing a performance drop of up to 27%, while FW causes a performance drop of less than 10% in all experiments. To draw meaningful conclusions from these numbers, we need to understand what are the properties of a packet-processing flow that make it sensitive and/or aggressive with respect to contention.

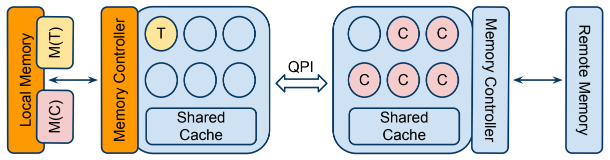
3.1 Contended Resources

We first identify which resources are responsible for contention. There are two candidates: the cache and the memory controller. We can rule out the processor interconnect, because our configuration ensures that each flow accesses its data locally, hence does not use this interconnect (Section 2.2).

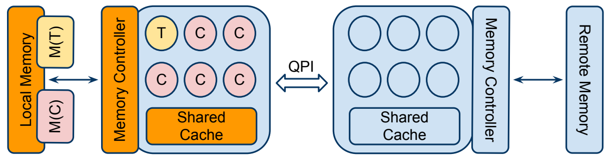
To assess the level of contention for the two candidate resources, we use the three system configurations illustrated in Figure 3. Each configuration allocates processing cores and memory to the co-running flows, so as to expose contention for different resources [7]: the first configuration creates contention only for the cache, the second one only for the memory controller, and the third one for both. In each configuration, we measure the contention-induced performance drop suffered by re-



(a) T contends with Cs only for the L3 cache. Cs' data is remote, hence accessed through a different memory controller.



(b) T contends with Cs only for the memory controller. Cs run on a different processor, hence use a different L3 cache.



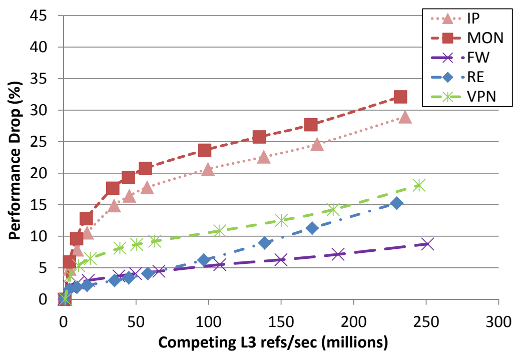
(c) T contends with Cs for both the memory controller and the L3 cache.

Figure 3: Configurations that expose contention for different resources. The resource that is contended in each configuration is highlighted. T denotes the target flow (whose performance drop we are measuring) and $M(T)$ denotes flow T's data structures. C denotes a competing flow and $M(C)$ denotes the corresponding data structures.

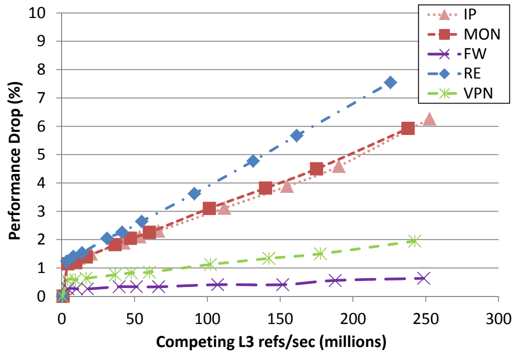
alistic flow types when they encounter different levels of competition. Figure 4 shows the drop suffered by each flow type when it co-runs with SYN flows, as a function of the cache refs/sec performed by the SYN flows.

These numbers show that the dominant contention factor is the cache. The most sensitive flow type (MON) suffers up to 32% when competing for the cache only (the curve with square data points in Figure 4(a)) and up to 6% when competing for the memory controller only (the curve with square data points in Figure 4(b)).

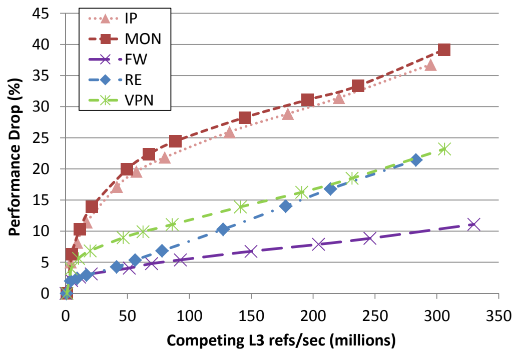
Our conclusion relates to prior work as follows: It differs from the conclusion drawn by running SPEC benchmarks on multicore platforms—in that case, the dominant contention factors were found to be the memory controller and the processor interconnect [7, 34]. The difference may come from the fact that packet-processing workloads benefit from the cache more than SPEC benchmarks and/or the fact that, in our context, overloading the interconnect is unnecessary. Our conclusion is consistent with recent results on software routers: in a software router running on an Intel Nehalem platform, as long as we have sufficient network I/O capacity, the bottleneck lies with the CPU and/or memory latency [16].



(a) Contention for the L3 cache. Performance drop suffered by each flow type in the configuration of Figure 3(a).



(b) Contention for the memory controller. Performance drop suffered by each flow type in the configuration of Figure 3(b).



(c) Contention for both resources. Performance drop suffered by each flow type in the configuration of Figure 3(c).

Figure 4: The effect of contention for different resources. For each realistic flow type X, we co-run a flow of type X with 5 flows of type SYN multiple times, ramping up the number of cache refs/sec performed by the SYN flows. We measure the performance drop suffered by the flow of type X as a function of the competing cache refs/sec.

3.2 Sensitivity and Aggressiveness

We now look at which properties of a packet-processing flow determine its sensitivity and aggressiveness, i.e., the amount of damage that it suffers from its co-runners and the amount of damage that it causes to them.

First, we observe a positive correlation between a flow's sensitivity to contention and the number of cache hits per second that it achieves during a solo run. Figure 2(b) shows that the higher the number of hits per second achieved by a flow type during a solo run (Table 1), the higher the average performance drop suffered by the flow. This makes sense: sharing a cache with co-runners causes memory references that would result in cache hits (if the flow ran alone) to become cache misses; the more hits per second a flow achieves during a solo run, the more opportunity there exists for these hits to become misses, leading to higher performance drop.

Second, we observe that the amount of damage suffered by a given flow is mostly determined by the number of competing cache refs/sec, not so much by the types of the competitors. Said differently, two flows, C_1 and C_2, that perform the same number of cache refs/sec will cause roughly the same performance drop to a given co-runner, regardless of whether C_1 and C_2 involve the same or different types of packet processing. This can be seen in Figure 5, which shows the performance drop suffered by different flows when they co-run with SYN as well as realistic competitors. For instance, a MON flow suffers a 27% drop when competing with 5 RE flows that generate 80 million cache refs/sec, and it suffers a 24% drop when competing with 5 SYN flows that generate the same number of cache refs/sec. So, RE flows cause about the same damage with SYN flows that generate the same rate of cache references, even though RE involves redundancy elimination, whereas SYN involves random memory accesses.

We found this observation partly intuitive and partly surprising: The intuitive part is that more competing cache references result in more damage, because they reduce the effective cache space of the target flow. The surprising part is that the particular memory access pattern of the competitors is not significant, and instead the rate of competing cache references mostly determines the amount of damage suffered by flows. It is worth noting that most of the complexity of existing mathematical models that predict contention effects comes from their effort to characterize the memory access patterns of the co-runners and their interaction.

Third, we observe that a sensitive flow's performance at first drops sharply with the number of competing cache refs/sec, however, beyond some point, the drop slows down significantly. For instance, as we see in Figure 5, a MON flow's performance drops by 20% when competition goes from 0 to 50 million cache refs/sec, but only an extra 5% when competition goes from 50 to 100 million cache refs/sec. As a result, a MON flow's performance drops roughly the same, whether it is co-running with IP, MON, or RE competitors, since all these flows contribute at least 50 million cache refs/sec.

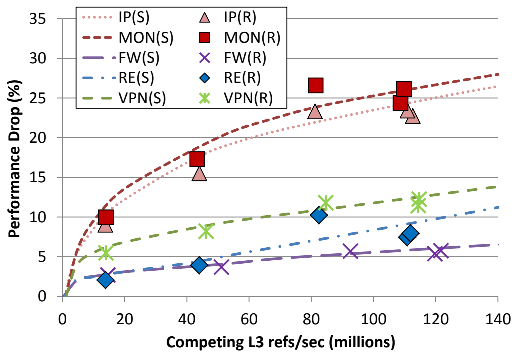


Figure 5: A merge of Figures 2(a) and 4(c). It shows the performance drop suffered by each flow type when it co-runs with SYN flows (curves) as well as realistic flows (individual points). For example, the curve MON(S) shows the performance drop suffered by a MON flow when it co-runs with SYN flows, while the individual squares MON(R) show the performance drop suffered by a MON flow when it co-runs with various realistic flow types. Each MON(R) square corresponds to a different realistic competitor type.

Summary. We made three observations: (a) A flow's sensitivity to resource contention depends on the number of hits/sec that the flow achieves during a solo run. (b) The specific amount of damage that a flow suffers due to contention is mostly determined by the number of cache refs/sec performed by its competitors, and not by the exact type of packet processing that they perform. (c) The performance of a sensitive flow at first drops sharply as the number of competing cache refs/sec increases; however, once a "turning point" is reached, the performance drop suffered by each sensitive flow stays within a relatively small range, no matter what type of co-runners it is competing with.

3.3 Explanation of our Observations

Before we use these observations, we provide intuition and potential explanations for them. Since the dominant contention factor is the cache, we concentrate on cache contention.

Sensitivity depends on cache hits/sec. We can express the performance drop suffered by a flow due to cache contention as follows:

- Suppose the flow achieves h cache hits/sec and processes n packets/sec during a solo run.
- Suppose that, due to contention, the flow suffers *hit-to-miss conversion rate* κ, i.e., each memory reference that was a hit during a solo run turns into a miss with probability κ.

- Without contention, processing n packets takes 1 second. With contention, processing n packets results in $\kappa \cdot h$ extra cache misses and takes $1 + \delta \cdot \kappa \cdot h$ seconds, where δ is the extra time needed to complete a memory reference that is a cache miss instead of a cache hit.

- Hence, the performance drop suffered by the flow in terms of packets/sec will be

$$\frac{n - \frac{n}{1+\delta\kappa h}}{n} = \frac{1}{1 + \frac{1}{\delta\kappa h}}. \qquad (1)$$

Performance drop increases with competition (for the cache), primarily because the hit-to-miss conversion rate increases with competition. The value of δ provided by our platform's specs is 43.75 nanoseconds—although, in practice, its exact value depends on the nature of memory accesses and also slowly increases with competition.

In the worst case, the hit-to-miss conversion rate is $\kappa = 1$, i.e., all of the cache hits achieved by the target flow during a solo run turn into misses due to contention. Figure 6 shows this worst-case performance drop as a function of the number of cache hits/sec achieved by the flow during a solo run, for different values of δ. E.g., assuming $\delta = 43.75$ nanoseconds, if a packet-processing flow achieves fewer than 20 million cache hits/sec during a solo run, even if all the hits turn into misses, the flow's performance cannot drop by more than 47%.

As a side note, according to Equation 1, a flow's worst-case performance drop depends only on the hits/sec achieved by the flow during a solo run, not by other characteristics of the flow (such as cycles spent on computation or total memory references per second); this is what makes hits/sec a good metric for a flow's worst-case sensitivity to contention.

Aggressiveness is determined by cache refs/sec. We observed that, in our setup, the aggressiveness of a set of flows is mostly determined by their cache refs/sec: a set of realistic flows and a set of SYN flows that perform the same number of cache refs/sec cause roughly the same damage to a given target flow T.

We explain this as follows: Each of our realistic packet-processing flows accesses at least a few megabytes of data. When several of these flows co-run, they access a total amount of data that is significantly larger than the cache size, which causes them to access the cache close to uniformly. As a result, from the point of view of a target flow T that shares the cache with these flows, they behave similarly to a set of SYN flows (that access the cache uniformly by construction).

In Section 6, we briefly discuss the scenario where the working-set sizes of the competing flows are relatively small (such that the cache is not saturated) and explain why we do not address that scenario in this paper.

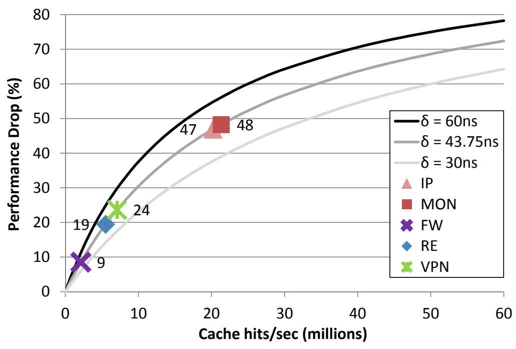


Figure 6: Estimated maximum performance drop suffered by a flow as a function of the cache hits/sec it achieves during a solo run. The estimates are based on Equation 1, for $\kappa = 1$ and different values of δ. The graph also shows the data points that correspond to our realistic packet-processing flows, assuming $\delta = 43.75$ nanoseconds. For example, the maximum performance drop that could be suffered by an IP flow is 47%.

Shape of the performance drop. To understand how performance drop changes with competition, we look at how the hit-to-miss conversion rate changes with competition. Figure 7 shows the conversion rate suffered by a MON flow as a function of cache competition, as we measure it on our platform (using the configuration of Figure 3(a)) and as we analytically estimate it using a simple model. We will use the model to provide intuition (*not* accurate prediction), then discuss how it matches the measured data.

We describe the model in our technical report [15] and summarize here the gist: We have a target flow T that shares a cache with a set of competitors.

- Consider a sequence of cache references, $\langle t, c_1, c_2, \ldots, c_Z, t' \rangle$, where: t and t' are two consecutive references performed by flow T to the same cache line, t' was a hit during a solo run, and $c_i, i = 1..Z$, are the competing references that occur between t and t'.

- Suppose that each competing reference c_i evicts the content cached by t with probability p_{ev}, independently from any other competing reference. t' is a hit if none of the Z competing references evict this content, i.e., $P(hit|Z) = (1 - p_{ev})^Z$. The target flow's hit-to-miss conversion rate is $1 - P(hit)$.

- Suppose that each reference that occurs after t is: either a competing reference, with probability p_c, or t', with probability $p_t = 1 - p_c$. Hence, Z is a random variable of geometric distribution with success probability p_t, i.e., $P(Z = z) = (1 - p_t)^z p_t$.

To compute $P(hit)$ as a function of competition, we need to know how p_{ev} and p_t change with competition. The following assumptions allow us to approximate them: (a) the competitors access the cache uniformly, (b) the target flow accesses its data uniformly, and (c) the target flow and the competitors have similar sensitivity to contention, i.e., suffer approximately the same hit-to-miss conversion rate.

Figure 7 shows that the shape of a flow's conversion rate as a function of competition can be explained as the result of basic cache sharing: The model-derived curve has a shape similar to the empirically derived curve (sharp rise at first, significant slow-down beyond some point), even though the model provides basic probabilistic analysis of cache sharing without considering any special feature of our platform. Note that, if we plug the model-derived conversion-rate values from Figure 7 into Equation 1 (for the value of h that corresponds to a MON flow), we get an analytical estimate of a MON flow's performance drop as a function of competition, which also has a shape similar to the corresponding empirically derived curve.

Our model captures the shape, but overestimates the value of the conversion rate. This is because the model assumes that the target flow accesses its data uniformly, which is usually not the case. In Figure 7, we see that different MON functions are affected differently by contention: (a) "flow_statistics" suffers a conversion rate that is well captured by the model, which makes sense because the flow table is accessed uniformly. (b) "check_ip_header" and "skb_recycle" suffer insignificant conversion rates. Our explanation is that these functions reference the same few cacheable data with every received packet (e.g., book-keeping structures), so, their cacheable data is almost never evicted by their competitors. (c) "radix_ip_lookup" is somewhere in the middle. We think this is because the root of the radix trie and its children are "hot spots," i.e., they are accessed frequently enough to remain in the cache, whereas the rest of the trie does not have any such hot spots. However, for all functions, most of the hits that *are* susceptible to conversion are converted by the time competition reaches 50 million cache refs/sec—and our model does capture that effect.

As a side note, mathematical models that try to predict the effects of resource contention are complex because they try to analytically compute p_{ev} and p_t as a function of competition, and this is a hard task. Suppose the target flow performs r_t cache refs/sec during a solo run. The competitors cause it to suffer extra misses that "slow it down," i.e., cause it to perform fewer than r_t cache refs/sec; how much fewer depends on the competitors' cache refs/sec. At the same time, the target flow slows down the competitors by a degree that depends on the target flow's cache refs/sec. In the end, the relative

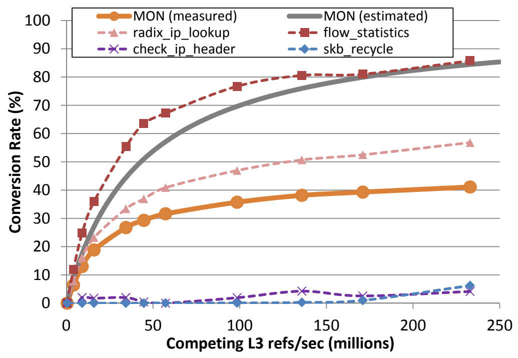


Figure 7: Measured and estimated hit-to-miss conversion rate suffered by a MON flow that shares the cache with SYN competitors, as a function of the competing cache refs/sec. The graph also shows the measured conversion rate suffered by separate functions of the MON flow. "flow_statistics" performs all the NetFlow-specific processing. "radix_ip_lookup" performs IP-table lookups. "check_ip_header" checks whether each packet has a valid IP header. "skb_recycle" performs memory management.

frequency of target and competitor memory references (which directly affects p_t) is the result of a complex interaction among the co-runners' particular access patterns. We were able to side-step this complexity (and crudely approximate p_{ev} and p_t) because our goal was not to predict but merely to explain why increasing competition beyond some point does not significantly increase the resulting performance drop.

Summary. Our observations can be explained as the result of multiple processes sharing a last-level cache. This is not particular to our platform, but a universal artifact of modern server architectures.

4 Predicting Contention

In this section, we show how to accurately predict the overall and per-flow performance of our platform using simple profiling of each packet-processing flow running alone. Our prediction is based on the observation that a workload's aggressiveness is determined by the number of cache refs/sec that it performs, while it does not depend significantly on other workload properties.

Suppose we plan to co-run a flow T with $|C|$ competing flows $C_1, C_2, ...C_{|C|}$. We predict flow T's performance as follows:

1. We measure the number of last-level cache refs/sec r_i performed by each flow C_i during a solo run.

2. We co-run flow T with different SYN flows, ramping up the number of cache refs/sec performed by

the SYN flows. We plot flow T's performance drop as a function of the number of competing cache refs/sec.

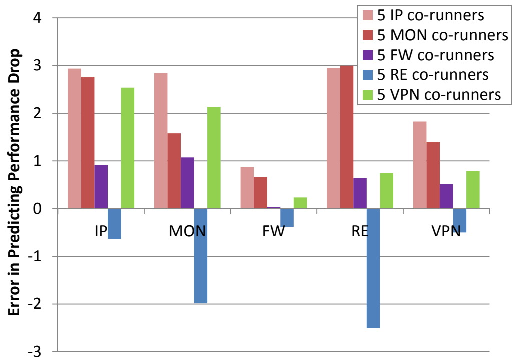
3. We predict that flow T's performance drop will be equal to the value of the plot (derived at step #2) that corresponds to $\sum_{i=1}^{|C|} r_i$ competing cache refs/sec.

We rely on two assumptions. First, we assume that the competing flows will affect the flow T as much as a set of SYN flows that perform the same number of cache refs/sec (this is well supported by the numbers in Figure 5). Second, we assume that each competing flow C_i will perform as many cache refs/sec as it does during a solo run.

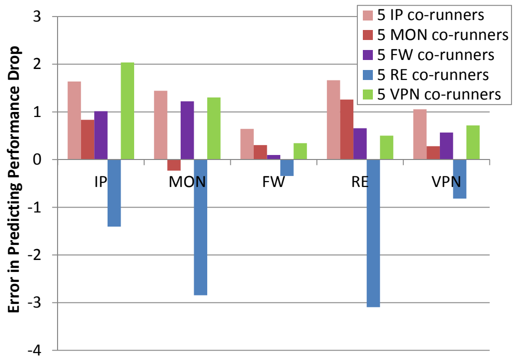
Our second assumption introduces a prediction error of a few percentage points: In reality, a competing flow C_i that belongs to a sensitive type (e.g., IP or MON) will also suffer due to contention, hence its processing will slow down, resulting in fewer cache refs/sec than it performs during a solo run. By assuming that each competing flow C_i will perform as many cache refs/sec as it does during a solo run, we overestimate the competition that flow T will encounter, hence underestimate its performance. However, the resulting prediction error is small, because of the shape of the performance drop that we observed in Section 3: Once the number of competing cache refs/sec exceeds 50 millions or so, small changes in the number of competing cache refs/sec do not significantly change the damage to a sensitive flow. And sensitive flows like IP or MON (the ones whose number of cache refs/sec we overestimate) are also aggressive flows, i.e., they push the number of competing refs/sec beyond the 50 million turning point.

To validate our prediction method, we first reuse the workloads introduced in the beginning of Section 3: for each possible pair of realistic flow types X and Y, we co-run a flow of type X with 5 flows of type Y. We have already seen the performance drop suffered by each flow type in each of these scenarios (Figure 2); we will now look at how well we can predict these performance drops and how much of our error is due to each assumption. Figure 8(a) shows our prediction error, i.e., the difference between predicted and actual performance drop suffered by each flow type in each scenario. Figure 8(b) shows what the error *would* be, if we knew the exact number of competing cache refs/sec (we refer to this scenario as "prediction assuming perfect knowledge of the competition"). Figure 8(c) shows the absolute difference between predicted and actual performance drop suffered by each flow type, averaged across all scenarios.

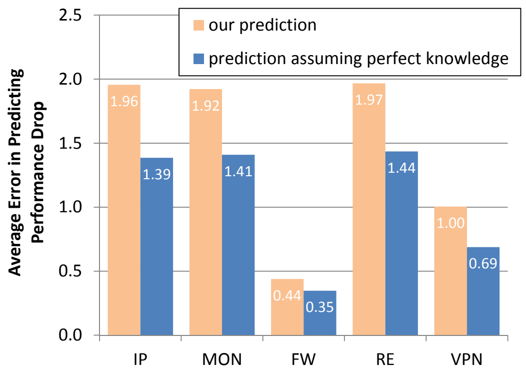
The average prediction error for each of the realistic flow types is less than 2% (tallest bars in Figure 8(c)). Our worst prediction errors are below 3%, and they correspond to the 2 leftmost bars in each group in Fig-



(a) Our prediction error. Difference between predicted and actual performance drop suffered by each flow type in each scenario.



(b) Prediction error assuming perfect knowledge of the competition. Difference between predicted and actual performance drop suffered by each flow type in each scenario, when we have perfect knowledge of the competing cache refs/sec.



(c) Average prediction error. Absolute difference between predicted and actual performance drop suffered by each flow type, averaged across all 5 scenarios that involve a target flow of that type. For example, we predict the performance drop suffered by a MON flow with an average error of 1.92% across all 5 scenarios (that involve a target MON flow).

Figure 8: Prediction errors for workloads with 2 flow types. For each pair of realistic flow types X and Y, we run an experiment in which a flow of type X co-runs with 5 flows of type Y. We measure/predict the performance drop suffered by the flow of type X.

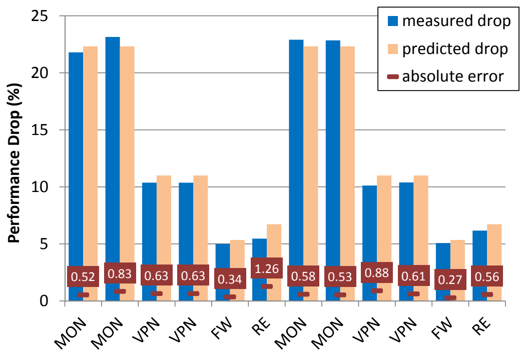


Figure 9: Prediction errors for a mixed workload. Predicted and actual performance drop suffered by each flow (and the absolute difference between the two).

ure 8(a): realistic flows that co-run with 5 IP or 5 MON competitors, respectively. In these scenarios, we overestimate the performance drop suffered by the target flow, partly because we assume that its co-runners will perform as many cache refs/sec as in the solo run. Actually, IP and MON are sensitive flow types that do suffer because of contention and perform fewer cache refs/sec compared to the solo run. The difference between the corresponding bars in Figure 8(a) and Figure 8(b) represents the error introduced by our second assumption. The rest of the error is due to our first assumption that the co-runners cause as much damage as a set of SYN flows that perform the same number of cache refs/sec.

We also validate our prediction method using a mixed workload: 2 MON, 2 VPN, 1 FW, and 1 RE flow per processor. Figure 9 shows the actual and predicted performance drop suffered by each flow, as well as the difference between the two. This time, we predict the performance drop suffered by each flow in the mix with a maximum error of 1.26%.

Containing hidden aggressiveness. Our prediction relies on offline profiling, i.e., running each packet-processing flow alone and measuring certain properties. However, it is possible that a flow (accidentally or on purpose) exhibits different behavior during offline profiling than during the actual run—a contrived example would be a flow that normally performs FW processing (i.e., is not aggressive), but, once it receives a specially crafted packet (potentially from an attacker), it switches mode and performs SYN_MAX processing (i.e., becomes very aggressive). Such a flow could mislead the system administrator into expecting significantly higher performance from her system and under-provisioning the system accordingly.

Nevertheless, a practical implication of our results is that an administrator can control the aggressiveness of each packet-processing flow simply by throttling the flow's rate of memory accesses. To verify this, we add to the beginning of each flow a "control element," which performs a configurable number of simple CPU operations, with the purpose of "slowing down" the flow and controlling the rate at which it performs memory accesses. At the same time, we monitor the rate at which each flow performs memory accesses using hardware performance counters and, if a flow exceeds the rate exhibited during its offline profiling, we configure its control element to slow it down accordingly.

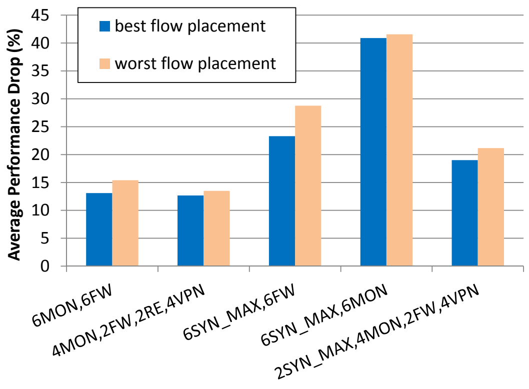
We tested this simple technique on our system and found that it ensures that each packet-processing flow performs no more than the profiled number of cache refs/sec. Thus, it is practical for an administrator to contain undue aggressiveness and achieve predictable performance.

5 Minimizing Contention via Scheduling

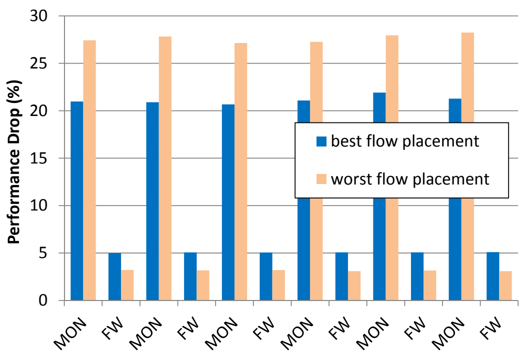
In this section, we explore the potential benefit of contention-aware scheduling [34] for packet-processing platforms. This is a family of techniques that solve the following problem: given J processing jobs and a multicore platform with J cores, how should we assign jobs to cores to minimize resource contention between the jobs and maximize the platform's overall performance? The basic idea at the core of the proposed solutions is to profile (offline or real-time) each process and avoid co-running aggressive with sensitive processing jobs.

To quantify the potential benefit of contention-aware scheduling for our system, we consider different combinations of 12 packet-processing flows. For each combination, we measure the contention-induced performance drop (averaged across all flows) under the worst and best flow-to-core placement (Figure 10(a)). The difference between these two numbers expresses the maximum we can gain in overall system performance through contention-aware scheduling.

For realistic-flow combinations, the maximum we can gain in overall system performance is 2% (Figure 10(a)). The flow combination for which we gain this maximum benefit is 6 MON and 6 FW flows. Figure 10(b) shows the per-flow performance drop for this combination, under the worst and best placement. The worst placement assigns the 6 MON flows to one processor and the 6 FW flows to the other, such that all the 6 MON flows (which are both aggressive and sensitive) have to compete with each other for the L3 cache; this causes a performance drop of 27% to each MON flow and an overall system performance drop of 15%. The best placement is the one that assigns 3 MON and 3 FW flows to each processor, such that each MON flow has to compete with only 2 other MON flows for the L3 cache; this causes a performance drop of 21% to each MON flow and an overall



(a) Average per-flow performance drop suffered under the worst and best placement, for different flow combinations.



(b) Per-flow performance drop suffered under the worst and best placement, for the 6-MON/6-FW combination.

Figure 10: Benefit of contention-aware scheduling. Performance drop suffered under the worst and best flow-to-core placement.

system performance drop of 13%. Hence, the extra damage introduced by the worst versus the best placement is 6% for each MON flow and 2% for the overall system.

Of all the possible realistic-flow combinations (given the flows that we implemented), this particular combination (6 MON and 6 FW flows) allows for the biggest overall improvement, because it is an equal mix of the most and least sensitive/aggressive flow types. One may think, at first, that a combination of more aggressive and/or sensitive flows (e.g., replacing the FW flows with IP or RE flows) would allow for a bigger improvement, but that is not the case: To create as big a difference as possible between the worst and best placement, we need a mix of sensitive, aggressive, and non-aggressive flows, such that in the worst placement sensitive flows co-run with the aggressive ones, whereas in the best placement sensitive flows co-run with the non-aggressive ones. Indeed, any other realistic-flow combination that we tried yielded an even smaller difference between worst and best placement.

This lack of (significant) difference between the worst and best placement makes sense, if we consider the observations in Section 3.2: once the competing cache refs/sec reach 50 millions or so, the performance drop suffered by a sensitive flow stays within a relatively small range, no matter which particular flows it co-runs with. Consider the 6-MON/6-FW combination: under the worst placement, each MON flow competes with 5 other MON flows, which generate about 100 million competing refs/sec, which causes the MON flow to suffer a performance drop of 27%; under the best placement, each MON flow competes with 2 other MON flows plus 3 FW flows, which generate about 60 million refs/second, which causes the MON flow to suffer a performance drop of 21%. In the end, as long as a placement generates more than a few tens of millions of cache refs/sec, it causes more or less the same performance drop to each sensitive flow.

If we consider non-realistic flows, the maximum we can gain in overall performance is 6%, for the 6 SYN_MAX, 6 FW combination (Figure 10(a)). SYN_MAX is the most aggressive and at the same time the most sensitive flow that we were able to craft (recall that it performs no processing other than memory accesses at the highest rate possible). So, even in the scenario where we have an equal mix of flows manifesting the most aggressive/sensitive behavior that we were able to generate in our system (SYN_MAX) and non-aggressive/non-sensitive flows (FW), the maximum benefit of contention-aware scheduling with respect to overall performance is 6%. Any other combination that we tried yielded an even smaller benefit.

6 Discussion

All the scenarios we considered have two common characteristics: each core runs a single packet-processing flow (Section 2.2) and the aggregate working-set size of the competing flows far exceeds the size of the cache (Section 3.3). If each core runs multiple flows, these compete for the L1 and L2 caches, so considering only the L3 accesses may not be sufficient to predict performance drop. If the working-set sizes of the flows are close to their fair share of the cache, then considering only the competing cache refs/sec may not be sufficient to characterize a workload's aggressiveness. These conditions may occur, for instance, in an active-networking setting, where large numbers of end users instantiate many small packet-processing flows on intermediate network elements.

We focused on one-flow-per-core, saturated-cache scenarios because we think that these are most likely to occur in the near future: State-of-the-art general-purpose platforms already offer tens of cores, and we consider it unlikely that a network operator would need to support more than a few tens of different packet-processing types. Moreover, the point of building programmable packet-processing platforms is to make it easy to deploy new, interesting types of packet processing. All the emerging types of packet processing that we are aware of (e.g., redundancy elimination, deep packet inspection, application acceleration) would require several megabytes of frequently accessed data in a realistic network setting (e.g., a network interface that handles a few gigabits per second, located on the border of an Internet Service Provider). In state-of-the-art platforms, the size of the shared last-level cache is less than 3MB per core (and this will not increase in the near future, if the current architecture trends persist). Hence, we expect that running any combination of interesting packet-processing applications on a state-of-the-art multicore platform would saturate the shared caches.

7 Related Work

In recent years, we have seen a renewed interest in general-purpose networking, both by the industry [4] and the research community. Several research prototypes have demonstrated that general-purpose hardware is capable of high-performance packet processing (line rates of 10 Gbps or more), assuming simple, uniform workloads, where all the packets are subjected to one particular type of packet processing: IP forwarding [16], GPU-aided IP forwarding [17], multi-dimensional packet classification [23], or cryptographic operations [18]. Like all this work, our ultimate goal is to build high-performance software packet-processing systems. However, our focus here is to show that such a system can achieve *predictable* performance while running a wide range of packet-processing applications and serving multiple clients with different needs.

Researchers have been working for more than two decades on mathematical models for predicting the effects of resource contention. In the eighties and nineties, this was pursued in the context of general-purpose systems with simultaneous multithreading [5, 28, 31]. In the last decade, the focus has shifted to general-purpose multicore systems with shared caches [10, 12, 29, 32]. Zhang et al. recently questioned the need for prediction, with the argument that cache contention does not significantly affect the performance of modern parallel applications (in particular, PARSEC benchmarks) [33]. We show that, in the context of packet processing, resource contention can cause significant performance drop (up to 27%), however, we can accurately predict that without mathematical modeling. We should note that modeling does not remove the need for application profiling: all proposed models require as input at least the stack distance profile [24] of each application, which requires either instruction-set simulation of the application, or binary instrumentation and program analysis of the application, or co-running the application with a set of synthetic benchmarks [32].

A complementary topic to contention prediction is contention-aware scheduling: how to assign processes to cores so as to maximize overall system performance [7, 13, 19, 20, 25, 34]. We show that, in the context of packet processing, contention-aware scheduling does not significantly improve overall performance.

Finally, our work falls under the broader effort of exploring how software systems should be architected to exploit multicore architectures. That work has typically focused on redesigning software to expose parallelism—most recently by eliminating serial execution bottlenecks [9]. In contrast, we focus on packet-processing workloads, which are already amenable to parallel execution. Given a seemingly perfectly parallel system like a software packet-processing platform, we analyze what are the challenges involved in running such a system and—as a first step—what we can do to make its performance predictable.

8 Conclusion

We presented a software packet-processing system that combines ease of programmability with predictable performance, while supporting a diverse set of packet-processing flows. We showed that, in our system, we can accurately predict the contention-induced performance drop suffered by each flow (with an error smaller than 3%) thanks to two key observations: First, the performance drop suffered by a given flow is mostly determined by the number of cache references per second performed by its competitors, and not by the exact type of packet processing that they perform. Second, as long as the number of competing cache references per second exceeds a certain threshold, the performance drop suffered by a sensitive flow stays within a relatively small range, no matter what type of co-runners it is competing with. We also showed that, in our system, overall performance depends little on how different flows are scheduled on different cores, hence, contention-aware scheduling may not be worth the effort. We quantitatively argued that our results are not artifacts of a particular hardware architecture, rather they should hold on any modern multicore platform.

Acknowledgments. We would like to thank Aditya Akella, Ashok Anand, Michele Catasta, Jiaqing Du, Nikola Knezevic, Ovidiu Mara, Alexandra Olteanu, Simon Schubert, Vyas Sekar, our shepherd Srinivasan Seshan, and the anonymous reviewers for their help and constructive feedback.

References

[1] Cisco IOS NetFlow. `http://www.cisco.com/web/go/netflow`.

[2] Intel 82599 10 GbE Controller Datasheet. `http://download.intel.com/design/network/datashts/82599_datasheet.pdf`.

[3] OProfile. `http://oprofile.sourceforge.net`.

[4] Why Use Vyatta? `http://www.vyatta.org/getting-started/why-use`.

[5] A. Agarwal, M. Horowitz, and J. Hennesy. An Analytical Cache Model. *Transactions on Computer Systems (TOCS)*, 7:184–215, 1989.

[6] K. Argyraki, S. Baset, B.-G. Chun, K. Fall, G. Iannaccone, A. Knies, E. Kohler, M. Manesh, S. Nedevschi, and S. Ratnasamy. Can Software Routers Scale? In *Proceedings of the ACM SIGCOMM Workshop on Programmable Routers for Extensible Services of TOmorrow (PRESTO)*, 2008.

[7] S. Blagodurov, S. Zhuravlev, M. Dashti, and A. Fedorova. A Case for NUMA-Aware Contention Management on Multicore Processors. In *Proceedings of the USENIX Annual Technical Conference*, 2011.

[8] W. J. Bolosky and M. L. Scott. False Sharing and its Effect on Shared Memory Performance. In *Proceedings of the USENIX Symposium on Experiences with Distributed and Multiprocessor Systems*, 1993.

[9] S. Boyd-Wickizer, A. T. Clements, Y. Mao, A. Pesterev, M. F. Kaashoek, R. Morris, and N. Zeldovich. An Analysis of Linux Scalability to Many Cores. In *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.

[10] D. Chandra, F. Guo, S. Kim, and Y. Solihin. Predicting Inter-Thread Cache Contention on a Chip Multi-Processor Architecture. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, 2005.

[11] B. Chen and R. Morris. Flexible Control of Parallelism in a Multiprocesor PC Router. In *Proceedings of the USENIX Annual Technical Conference*, 2001.

[12] X. E. Chen and T. M. Aamodt. A First-Order Fine-Grained Multithreaded Throughput Model. In *Proceedings of the IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2009.

[13] G. Dhiman, G. Marchetti, and T. Rosing. vGreen: a System for Energy-Efficient Computing in Virtualized Environments. In *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED)*, 2009.

[14] M. Dobrescu, K. Argyraki, M. Manesh, G. Iannaccone, and S. Ratnasamy. Controlling Parallelism in Multi-core Software Routers. In *Proceedings of the ACM SIGCOMM Workshop on Programmable Routers for Extensible Services of TOmorrow (PRESTO)*, 2010.

[15] M. Dobrescu, K. Argyraki, and S. Ratnasamy. Toward Predictable Performance in Software Packet-Processing Platforms. Technical report, Ecole Polytechnique Fédérale de Lausanne (EPFL), Switzerland, 2012.

[16] M. Dobrescu, N. Egi, K. Argyraki, B.-G. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy. RouteBricks: Exploiting Parallelism to Scale Software Routers. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2009.

[17] S. Han, K. Jang, K. Park, and S. Moon. PacketShader: A GPU-accelerated Software Router. In *Proceedings of the ACM SIGCOMM Conference*, 2010.

[18] K. Jang, S. Han, S. Han, S. Moon, and K. Park. SSLShader: Cheap SSL Acceleration with Commodity Processors. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2011.

[19] Y. Jiang, X. Shen, J. Chen, and R. Tripathi. Analysis and Approximation of Optimal Co-Scheduling on Chip Multiprocessors. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2008.

[20] R. Knauerhase, P. Brett, B. Hohlt, T. Li, and S. Hahn. Using OS Observations to Improve Performance in Multicore Systems. *IEEE Micro*, 28:54–66, 2008.

[21] E. Kohler, R. Morris, B. Chen, J. Jannoti, and M. F. Kaashoek. The Click Modular Router. *ACM Transactions on Computer Systems (TOCS)*, 18(3):263–297, 2000.

[22] G. Lu, C. Guo, Y. Li, Z. Zhou, T. Yuan, H. Wu, Y. Xiong, R. Gao, and Y. Zhang. ServerSwitch: A Programmable and High Performance Platform for Data Center Networks. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2011.

[23] Y. Ma, S. Banerjee, S. Lu, and C. Estan. Leveraging Parallelism for Multi-dimensional Packet Classification on Software Routers. In *Proceedings of the ACM SIGMETRICS Conference*, 2010.

[24] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation Techniques for Storage Hierarchies. *IBM Systems Journal*, 9:78–17, 1970.

[25] A. Merkel, J. Stoess, and F. Bellosa. Resource-Conscious Scheduling for Energy Efficiency on Multicore Processors. In *Proceedings of the EuroSys Conference*, 2010.

[26] V. Sekar, N. Egi, S. Ratnasamy, M. K. Reiter, and G. Shi. Design and Implementation of a Consolidated Middlebox Architecture. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2012.

[27] N. T. Spring and D. Wetherall. A Protocol Independent Technique for Eliminating Redundant Network Traffic. In *Proceedings of the ACM SIGCOMM Conference*, 2000.

[28] G. E. Suh, S. Devadas, and L. Rudolph. Analytical Cache Models with Applications to Cache Partitioning. In *Proceedings of the International Conference on Supercomputing (ICS)*, 2005.

[29] D. Tam, R. Azimi, L. B. Soares, and M. Stumm. Rapidmrc: Approximating L2 Miss Rate Curves on Commodity Systems for Online Optimizations. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2009.

[30] L. Tang, J. Mars, N. Vachharajani, R. Hundt, and M. L. Soffa. The Impact of Memory Subsystem Resource Sharing on Datacenter Applications. In *Proceedings of the Annual International Symposium on Computer Architecture (ISCA)*, 2011.

[31] D. Thiebaud and H. S. Stone. Footprints in the Cache. *Transactions on Computer Systems (TOCS)*, 5:305–329, 1987.

[32] C. Xu, X. Chen, R. P. Dick, and Z. M. Mao. Cache Contention and Application Performance Prediction for Multi-Core Systems. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS)*, 2010.

[33] E. Z. Zhang, Y. Jiang, and X. Shen. Does Cache Sharing on Modern CMP Matter to the Performance of Comtemporary Multithreaded Programs? In *Proceedings of the ACM Symposium on the Principles and Practice of Parallel Programming (PPoPP)*, 2010.

[34] S. Zhuravlev, S. Blagodurov, and A. Fedorova. Addressing Shared Resource Contention in Multicore Processors via Scheduling. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2010.

Detecting and Defending Against Third-Party Tracking on the Web

Franziska Roesner, Tadayoshi Kohno, and David Wetherall
University of Washington

Abstract

While third-party tracking on the web has garnered much attention, its workings remain poorly understood. Our goal is to dissect how mainstream web tracking occurs in the wild. We develop a client-side method for detecting and classifying five kinds of third-party trackers based on how they manipulate browser state. We run our detection system while browsing the web and observe a rich ecosystem, with over 500 unique trackers in our measurements alone. We find that most commercial pages are tracked by multiple parties, trackers vary widely in their coverage with a small number being widely deployed, and many trackers exhibit a combination of tracking behaviors. Based on web search traces taken from AOL data, we estimate that several trackers can each capture more than 20% of a user's browsing behavior. We further assess the impact of defenses on tracking and find that no existing browser mechanisms prevent tracking by social media sites via widgets while still allowing those widgets to achieve their utility goals, which leads us to develop a new defense. To the best of our knowledge, our work is the most complete study of web tracking to date.

1 Introduction

Web tracking, the practice by which websites identify, and collect information about users — generally in the form of some subset of web browsing history — has become a topic of increased public debate. To date, however, the research community's knowledge of web tracking is piecemeal. There are many specific ways that identifying information might be gleaned (e.g., browser fingerprinting [4], ETags [2], and Flash cookies [21]) but little assessment of how tracking is integrated with web browsing in practice. Further complicating the situation is that the capabilities of different trackers depend strongly on their implementation. For instance, it is common for trackers like Google Analytics and Doubleclick to be mentioned in the same context (e.g., [14]) even though the former is implemented so that it cannot use unique identifiers to track users across sites while the latter can.

As the tracking arms race continues, the design of future web systems must be informed by an understanding of how web trackers retask browser mechanisms for tracking purposes. Our goal is thus to provide a comprehensive assessment of third-party tracking on the web today, where a third-party tracker is defined as a website (like `doubleclick.net`) that has its tracking code included or embedded in another site (like `cnn.com`). We focus on third-party tracking because of its potential concern to users, who may be surprised that a party with which they may or may not have chosen to interact is recording their online behavior in unexpected ways. We also explicitly focus on mainstream web tracking that uses cookies and other conventional local storage mechanisms (HTML5, Flash cookies) to compile records of users and user behavior. This is the most prevalent form of tracking today. More esoteric forms of tracking do exist — such as tracking with Etags [2], visited link coloring [10], and via the cache — and would threaten privacy if widely deployed, but they are not commonly used today. We similarly exclude inference-based browser and machine fingerprinting techniques, commonly used for online fraud detection [22, 24], in favor of explicit tracking that pinpoints users with browser state.

Our approach is to detect tracking as it is observed by clients; we achieve this goal by integrating web tracking detection directly into the browser. We began by looking at how real tracker code interacts with browsers, and from there distill five distinct behavior types. In our system, we are able to distinguish these different sets of behaviors, for example classifying Google Analytics and Doubleclick as distinct.

We then developed a Firefox browser extension to measure the prevalence of different web trackers and tracking behaviors. We aimed our tool at the 500 most popular and 500 less popular websites according to the Alexa rankings, as well as real user workloads as approximated with web traces generated from publicly available AOL search logs. Our measurements reveal extensive tracking. Pages are commonly watched by more than one of the over 500 unique trackers we found. These trackers exhibit a variety of nondeterministic behaviors, including hierarchies in which one tracker hands off to another. Several trackers have sufficient penetration that they may capture a large fraction of a user's browsing activity.

Our method also allowed us to assess how today's defenses reduce web tracking. We found that popup blocking, third-party cookie blocking and the Do Not Track header thwarted a large portion of cookie-based tracking without impacting functionality in most browsers, with the exception of tracking by social media widgets. Disabling JavaScript is more effective but can significantly impact the browsing experience. Tracking by social media widgets (e.g., Facebook) has rapidly grown in coverage and highlights how unanticipated combinations of browser mechanisms can have unexpected effects. Informed by our understanding of this kind of tracking, as well as the inadequacy of existing solutions, we developed the ShareMeNot extension to successfully defend against it while still allowing users to interact with the widgets.

To summarize, we make several contributions. Our classification of tracking behaviors is new, and goes beyond simple notions of first- and third-party tracking. Our measurements of deployed web trackers and how much they track users give the most detailed account of which we are aware to date of tracking in the wild, as well as an assessment of the efficacy of common defenses. Finally, our ShareMeNot extension provides a new defense against a practical threat. We now turn to additional background information in Section 2.

2 Background

Third-party web tracking refers to the practice by which an entity (the tracker), other than the website directly visited by the user (the site), tracks or assists in tracking the user's visit to the site. For instance, if a user visits `cnn.com`, a third-party tracker like `doubleclick.net` — embedded by `cnn.com` to provide, for example, targeted advertising — can log the user's visit to `cnn.com`. For most types of third-party tracking, the tracker will be able to link the user's visit to `cnn.com` with the user's visit to other sites on which the tracker is also embedded. We refer to the resulting set of sites as the tracker's *browsing profile* for that user. Before diving into the mechanisms of third-party tracking, we briefly review necessary web-related background.

2.1 Web-Related Background

Page Fetching. When a page is fetched by the browser, an HTTP request is made to the site for a URL in a new top-level execution context for that site (that corresponds to a user-visible window with a site title). The HTTP response contains resources of several kinds (HTML, scripts, images, stylesheets, and others) that are processed for display and which may trigger HTTP requests for additional resources. This process continues recursively until loading is complete.

Execution Context. A website can embed content from another domain in two ways. The first is the inclusion of an iframe, which delegates a portion of the screen to the domain from which the iframe is sourced — this is considered the *third-party domain*. The *same-origin policy* ensures that content from the two domains is isolated: any scripts running in the iframe run in the context of the third-party domain. By contrast, when a page includes a script from another domain (using `<script src=...>`), that script runs in the domain of the embedding page (the *first-party domain*), not in that of the script's source.

Client-Side Storage. Web tracking relies fundamentally on a website's ability to store state on the user's machine — as do most functions of today's web. Client-side state may take many forms, most commonly traditional browser cookies. A *cookie* is a triple (domain, key, value) that is stored in the browser across page visits, where domain is a web site, and key and value are opaque identifiers. Cookies that are set by the domain that the user visits directly (the domain displayed in the browser's address bar) are known as *first-party cookies*; cookies that are set by some other domain embedded in the top-level page are *third-party cookies*.

Cookies are set either by scripts running in the page using an API call, or by HTTP responses that include a Set-Cookie header. The browser automatically attaches cookies for a domain to outgoing HTTP requests to that domain, using Cookie headers. Cookies may also be retrieved using an API call by scripts running in the page and then sent via any channel, such as part of an HTTP request (e.g., as part of the URL). The same-origin policy ensures that cookies (and other client-side state) set by one domain cannot be directly accessed by another domain.

Users may choose to block cookies via their browser's settings menu. Blocking all cookies is uncommon[1], as it makes today's web almost unusable (e.g., the user cannot log into any account), but blocking third-party cookies is commonly recommended as a first line of defense against third-party tracking.

In addition to traditional cookies, HTML5 introduced new client-side storage mechanisms for browsers. In particular, *LocalStorage* provides a persistent storage area that sites can access with API calls, isolated by the same-origin policy. Plugins like Flash are another mechanism by which websites can store data on the user's machine. In the case of Flash, websites can set *Local Storage Objects* (LSOs, also referred to as "Flash cookies") on the user's file system.

2.2 Background on Tracking

Web tracking is highly prevalent on the web today. From the perspective of website owners and of trackers, it provides desirable functionality, including personalization, site analytics, and targeted advertising. A recent study [6] claims that the negative economic impact of preventing targeted advertising — or the underlying tracking mechanisms that enable it — is significant. From the perspective of a tracker, the larger a browsing profile it can gather about a user, the better service it can provide to its customers (the embedding websites) and to the user herself (e.g., in the form of personalization).

[1]On October 3, 2011, the Gibson Research Corporation cookie statistics page (`http://www.grc.com/cookies/stats.htm`) showed that almost 100% of 70,834 unique visitors in the previous week had first-party cookies enabled.

From the perspective of users, however, larger browsing profiles spell greater loss of privacy. A user may not, for instance, wish to link the articles he or she views on a news site with the type of adult sites he or she visits, much less reveal this information to an unknown third party. Even if the user is not worried about the particular third party, this data may later be revealed to unanticipated parties through court orders or subpoenas.

Despite the prevalence of tracking and the resulting public and media outcry — primarily in the United States and in Europe — there is a lack of clarity about how tracking works, how widespread the practice is, and the scope of the browsing profiles that trackers can collect about users. Tracking is often invisible; tools like the Ghostery Firefox add-on[2] aim to provide users with insight into the trackers they encounter on the web. What these tools do not consider, however, are the differences between types of trackers, their capabilities, and the resulting scope of the browsing profiles they can compile. For example, Google Analytics is commonly considered to be one of the most prominent trackers. However, it does not have the ability to create cross-site browsing profiles using the unique identifiers in its cookies. Thus, its prevalence is not correlated with the size of the browsing profiles it creates.

Storage and Communication. Our study focuses on *explicit* tracking mechanisms — tracking mechanisms that use assigned, unique identifiers per user — rather than *inferred* tracking based on browser and machine fingerprinting. Other work [25] has studied the use of fingerprinting to pinpoint a host with high accuracy. More specifically, all trackers we consider have two key capabilities:

1. The ability to store a pseudonym (unique identifier) on the user's machine.
2. The ability to communicate that pseudonym, as well as visited sites, back to the tracker's domain.

The pseudonym may be stored using any of the client-side storage mechanisms described in Section 2.1 — in a conventional browser cookie, in HTML5 LocalStorage, and in Flash LSOs, as well as in more exotic locations such as in ETags. There are multiple ways in which the browser may communicate information about the visited site to the tracker, e.g., implicitly via the *HTTP Referrer header* or explicitly via tracker-provided JavaScript code that directly transmits the results of an `document.referrer` API call. In some cases, a script running within a page might even communicate the visited page information in the GET or POST parameters of a request to a tracker's domain. For example, a tracker embedded on a site might access its own cookie and the referring page, and then pass this information on to another tracker with a URL of the form `http://tracker2.com/track?cookie_value=123&site=site.com`.

[2]`http://www.ghostery.com`

Different Scales of Tracking. Depending on the behaviors exhibited and mechanisms used by a tracker, the browsing profiles it compiles can be *within-site* or *cross-site*. Within-site browsing profiles link the user's browsing activity on one site with his or her other activity only on that site, including repeat visits and how the website is traversed, but not to visits to any other site. Cross-site browsing profiles link visits to multiple different websites to a given user (identified by a unique identifier or linked by another technique [16, 25]).

Behavioral Methodology. In this paper, we consider tracking behavior that is observable from the client, that is, from the user's browser. Thus, we do not distinguish between "can track" and "does track" — that is, we analyze trackers according to the capabilities granted by the behaviors we observe and not, for example, the privacy policies of the tracking sites.

From the background that we have introduced in this section, we step back and consider, via archetypical examples, the set of properties exhibited by trackers (Section 3.1); from these properties we formulate a classification of tracking behavior in Section 3.2.

3 Classifying Web Tracking Behavior

All web trackers that use unique identifiers are often bundled into the same category. However, in actuality diverse mechanisms are used by trackers, resulting in fundamentally different tracking capabilities. Our observations, based both on manual investigations and automated measurements, lead us to believe that it is incorrect to bundle together different classes of trackers — for example, Google Analytics is a within-site tracker, while Doubleclick is a cross-site tracker. To rigorously evaluate the tracking ecosystem, we need a framework for differentiating different tracker types. We develop such a framework here (Section 3.2). To inform this framework, we first dive deeply into an investigatory analysis of how tracking occurs today (Section 3.1), where we identify different properties of different trackers. We use our resulting framework as the basis for our measurements in Section 4.

3.1 Investigating Tracking Properties

In order to understand patterns of tracking behavior, we must first understand the properties of different trackers. We present several archetypal tracking examples here and, from each, extract a set of core properties.

Throughout this discussion, we will refer to cookies set under a tracker's domain as *tracker-owned* cookies. We introduce this term rather than using "third-party cookies" because a given cookie can be considered a first-party or a third-party cookie depending on the current browsing context. (For example, Facebook's cookie is a first-party cookie when the user visits `facebook.com`, but it is a third-party cookie when a Facebook "Like" button is

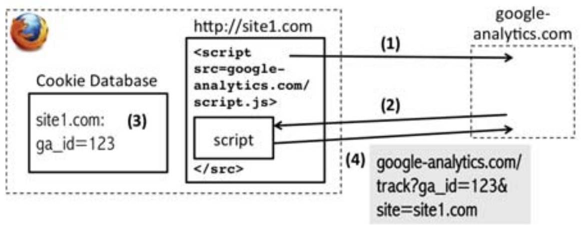


Figure 1: *Case Study: Third-Party Analytics.* Websites commonly use third-party analytics engines like Google Analytics (GA) to track visitors. This process involves (1) the website embedding the GA script, which, after (2) loading in the user's browser, (3) sets a site-owned cookie. This cookie is (4) communicated back to GA along with other tracking information.

embedded on another webpage.) Similarly, a cookie set under the domain of the website embedding a tracker is a *site-owned* cookie.

3.1.1 Third-Party Analytics

For websites that wish to analyze traffic, it has become common to use a third-party analytics engine such as Google Analytics (GA) in lieu of collecting the data and performing the analysis themselves. The webpage directly visited by the user includes a library (in the form of a script) provided by the analytics engine on pages on which it wishes to track users (see Figure 1).

To track repeat visitors, the GA script sets a cookie on the user's browser that contains a unique identifier. Since the script runs in the page's own context, the resulting cookie is site-owned, not tracker-owned. The GA script transfers this identifier to `google-analytics.com` by making explicit requests that include custom parameters in the URL containing information like the embedding site, the user identifier (from the cookie), and system information (operating system, browser, screen resolution, geographic information, etc.).

Because the identifying cookie is site-owned, identifiers set by Google Analytics across different sites are different. Thus, the user will be associated with a different pseudonym on the two sites, limiting Google Analytics's ability to create a cross-site browsing profile for that user.

Tracker Properties. We extract the following set of properties defining trackers like Google Analytics:

1. The tracker's script, running in the context of the site, sets a site-owned cookie.
2. The tracker's script explicitly leaks the site-owned cookie in the parameters of a request to the tracker's domain, circumventing the same-origin policy.

3.1.2 Third-Party Advertising

The type of tracking most commonly understood under "third-party tracking" is tracking for the purpose of targeted advertising. As an example of this type of tracking

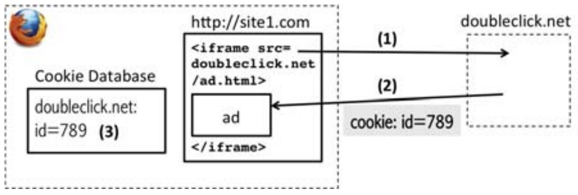


Figure 2: *Case Study: Third-Party Advertising.* When a website (1) includes a third-party ad from an entity like Doubleclick, Doubleclick (2-3) sets a tracker-owned cookie on the user's browser. Subsequent requests to Doubleclick from any website will include that cookie, allowing it to track the user across those sites.

scenario, we consider Google's advertising network, Doubleclick. Figure 2 shows an overview of this scenario.

When a page like `site1.com` is rendered on the user's browser, Doubleclick's code will choose an ad to display on the page, e.g., as an image or as an iframe. This ad is hosted by `doubleclick.net`, not by the embedding page (`site1.com`). Thus, the cookie that is set as the result of this interaction (again containing a unique identifier for the user) is tracker-owned. As a result, the same unique identifier is associated with the user whenever any site embeds a Doubleclick ad, allowing Doubleclick to create a cross-site browsing profile for that user.

Tracker Properties. We extract the following properties defining trackers like Doubleclick:

1. The tracker sets a tracker-owned cookie, which is then automatically included with any requests to the tracker's domain.
2. The tracker-owned cookie is set by the tracker in a third-party position — that is, the user never visits the tracker's domain directly.

3.1.3 Third-Party Advertising with Popups

A commonly recommended first line of defense against third-party tracking like that done by Doubleclick is third-party cookie blocking. However, in most browsers, third-party cookie blocking applies only to the setting, not to the sending, of cookies (in Firefox, it applies to both). Thus, if a tracker is able to maneuver itself into a position from which it can set a first-party cookie, it can avoid the third-party cookie blocking defense entirely.

We observed this behavior from a number of trackers, such as `insightexpressai.com`, which opens a popup window when users visit `weather.com`. While popup windows have other benefits for advertising (e.g., better capturing a user's attention), they also put the tracker into a first-party position without the user's consent. From there, the tracker sets and reads first-party cookies, remaining unaffected by third-party cookie blocking.

Tracker Properties. We extract the following properties defining trackers like Insight Express:

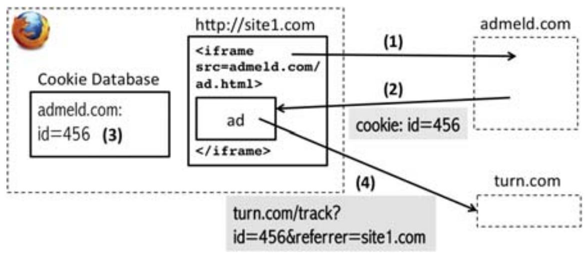


Figure 3: *Case Study: Advertising Networks.* As in the ordinary third-party advertising case, a website (1-2) embeds an ad from Admeld, which (3) sets a tracker-owned cookie. Admeld then (4) makes a request to another third-party advertiser, Turn, and passes its own tracker-owned cookie value and other tracking information to it. This allows Turn to track the user across sites on which Admeld makes this request, without needed to set its own tracker-owned state.

1. The tracker forces the user to visit its domain directly, e.g., with a popup or a redirect, allowing it to set its tracker-owned cookie from a first-party position.
2. The tracker sets a tracker-owned cookie, which is then automatically included with any requests to the tracker's domain when allowed by the browser.

3.1.4 Third-Party Advertising Networks

While, from our perspective, we have limited insights into the business models of third-party advertisers and other trackers, we can observe the effects of complex business relationships in the requests to third-parties made by the browser. In particular, trackers often cooperate, and it is insufficient to simply consider trackers in isolation.

As depicted in Figure 3, a website may embed one third-party tracker, which in turn serves as an aggregator for a number of other third-party trackers. We observed this behavior to be common among advertising networks. For example, `admeld.com` is often embedded by websites, and it makes further requests to trackers like `turn.com` and `invitemedia.com`. In these requests, `admeld.com` includes the information necessary to track the user, including the top-level page and the pseudonym from `admeld.com`'s own tracker-owned cookie. This means that `turn.com` does not need to set its own client-side state, but rather can rely entirely on `admeld.com`.

Tracker Properties. We extract the following properties defining trackers of this type:

1. The tracker is not embedded by the first-party website directly, but referred to by another tracker on that site.
2. The tracker relies on information passed to it in a request by the cooperating tracker.

3.1.5 Third-Party Social Widgets

An additional class of trackers doubles as sites that users otherwise visit intentionally, and often have an account with. Many of these sites, primarily social networking sites, expose social widgets like the Facebook "Like" button, the Twitter "tweet" button, the Google "+1" button

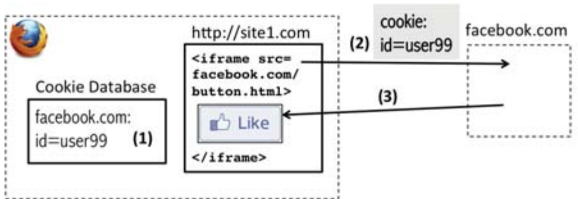


Figure 4: *Case Study: Social Widgets.* Social sites like Facebook, which users visit directly in other circumstances — allowing them to (1) set a cookie identifying the user — expose social widgets such as the "Like" button. When another website embeds such a button, the request to Facebook to render the button (2-3) includes Facebook's tracker-owned cookie. This allows Facebook to track the user across any site that embeds such a button.

and others. These widgets can be included by websites to allow users logged in to these social networking sites to like, tweet, or +1 the embedding webpage.

Figure 4 overviews the interaction between Facebook, a site embedding a "Like" button, and the user's browser. The requests made to `facebook.com` to render this button allow Facebook to track the user across sites just as Doubleclick can — though note that unlike Doubleclick, Facebook sets its tracker-owned cookie from a first-party position when the user voluntarily visits `facebook.com`.

Tracker Properties. We extract the following set of properties, where the important distinction to the Doubleclick scenario is the second property:

1. The tracker makes use of a tracker-owned cookies.
2. The user voluntarily visits the tracker's domain directly, allowing it to set the tracker-owned cookie from a first-party position.

3.2 A Classification Framework

We now present a classification framework for web trackers based on observable behaviors. This is in contrast to past work that considered business relationships between trackers and the embedding website rather than observable behaviors [9] and past work that categorized trackers based on prevalence rather than user browsing profile size, thereby commingling within-site and cross-site tracking [14]. In particular, from our manual investigations we distilled five tracking behavior types; we summarize these behaviors below and in Table 1. Table 2 captures the key properties from Section 3.1 and their relationships to these behavioral categories. In order to fall into a particular behavior category, the tracker *must* exhibit (at least) all of the properties indicated for that category in Table 2. A single tracker may exhibit more than one of these behaviors, as we discuss in more detail below.

1. **Behavior A (Analytics)**: The tracker serves as a third-party analytics engine for sites. It can only track users within sites.
2. **Behavior B (Vanilla)**: The tracker uses third-party storage that it can get and set only from a third-party

Category	Name	Profile Scope	Summary	Example	Visit Directly?
A	Analytics	Within-Site	Serves as third-party analytics engine for sites.	Google Analytics	No
B	Vanilla	Cross-Site	Uses third-party storage to track users across sites.	Doubleclick	No
C	Forced	Cross-Site	Forces user to visit directly (e.g., via popup or redirect).	InsightExpress	Yes (forced)
D	Referred	Cross-Site	Relies on a B, C, or E tracker to leak unique identifiers.	Invite Media	No
E	Personal	Cross-Site	Visited directly by the user in other contexts.	Facebook	Yes

Table 1: *Classification of Tracking Behavior.* Trackers may exhibit multiple behaviors at once, with the exception of Behaviors B and E, which depend fundamentally on a user's browsing behavior: either the user visits the tracker's site directly or not.

	Behavior				
Property	*A*	*B*	*C*	*D*	*E*
Tracker sets site-owned (first-party) state.	✓				
Request to tracker leaks site-owned state.	✓				
Third-party request to tracker includes tracker-owned state.		✓	✓		✓
Tracker sets its state from third-party position; user never directly visits tracker.		✓			
Tracker forces user to visit it directly.			✓		
Relies on request from another B, C, or E tracker (not from the site itself).				✓	
User voluntarily visits tracker directly.					✓

Table 2: *Tracking Behavior by Mechanism.* In order for a tracker to be classified as having a particular behavior (A, B, C, D, or E), it *must* display the indicated property. Note that a particular tracker may exhibit more than one of these behaviors at once.

position to track users across sites.

3. **Behavior C (Forced)**: The cross-site tracker forces users to visit its domain directly (e.g., popup, redirect), placing it in a first-party position.
4. **Behavior D (Referred)**: The tracker relies on a B, C, or E tracker to leak unique identifiers to it, rather than on its own client-side state, to track users across sites.
5. **Behavior E (Personal)**: The cross-site tracker is visited by the user directly in other contexts.

This classification is based entirely on tracker behavior that can be observed from the client side. Thus, it does not capture backend tracking behavior, such as correlating a user's browsing behavior using browser and machine fingerprinting techniques, or the backend exchange of data among trackers. Similarly, the effective type of a tracker encountered by a user depends on the user's own browsing behavior. In particular, the distinction between Behavior B and Behavior E depends on whether or not the user ever directly visits the tracker's domain.

Combining Behaviors. Most of these behaviors are not mutually exclusive, with the exception of Behavior B (Vanilla) and Behavior E (Personal) — either the user directly visits the tracker's domain at some point or not. That is, a given tracker can exhibit different behaviors on different sites or multiple behaviors on the same site. For example, a number of trackers — such as `quantserve.com` — act as both Behavior A (Analytics) and Behavior B (Vanilla) trackers. Thus, they provide site analytics to the embedding sites in addition to gathering cross-site browsing profiles (for the purposes of targeted advertising or additional analytics).

Through our analysis, we identified what was to us a surprising combination of behaviors — Behavior A (Analytics) and Behavior D (Referred) — by which a within-site tracker unintentionally gains cross-site

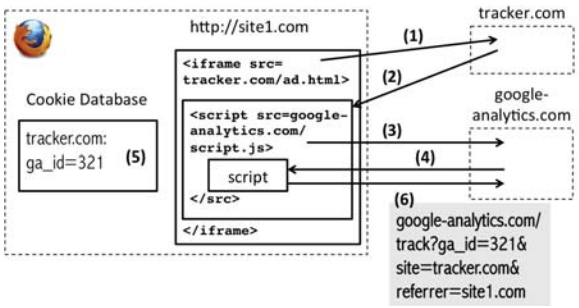


Figure 5: *Combining Behavior A and Behavior D.* When a Behavior A tracker like Google Analytics is embedded by another third-party tracker, rather than by the visited website itself, Behavior D emerges. The site-owned cookie that GA sets on `tracker.com` becomes a tracker-owned cookie when `tracker.com` is embedded on `site1.com`. The tracker then passes this identifier to Google Analytics, which gains the ability to track the user across all sites on which `tracker.com` is embedded.

tracking capabilities. We discovered this combination during our measurement study. For example, recall that Google Analytics is a within-site, not a cross-site, tracker (Behavior A). However, suppose that `tracker.com` uses Google Analytics for its own on-site analytics, thus receiving a site-owned cookie with a unique identifier. If `tracker.com` is further embedded on another site, this same cookie *becomes a tracker-owned cookie*, which is the same across all sites on which `tracker.com` is embedded. Now, when the usual request is made to `google-analytics.com` from `tracker.com` when it is embedded, *Google Analytics becomes a Behavior D tracker* — a cross-site tracker. Figure 5 shows an overview of this scenario. Note that we did not observe many instances of this in practice, but it is interesting to observe that within-site trackers can become cross-site trackers when different parties interact in complex ways. This observation is further evidence of the fact that the tracking ecosystem is complicated and that it is thus difficult to create simple, sweeping technical or policy solutions.

Robustness. We stress that this classification is agnostic of the practical manifestation of the mechanisms described above — that is, client-side storage may be done via cookies or any other mechanism, and information may be communicated back to the tracker in any way. This separation of semantics from mechanism makes the classification robust in the face of the evolution of specific client-side storage techniques used by trackers.

4 Detecting Trackers

Based on this classification framework, we created a tool — TrackingTracker — that automatically classifies trackers according to behavior observed on the client-side. TrackingTracker runs as a Firefox add-on, interposes on all HTTP(S) requests, and examines conventional cookies, HTML5 Local Storage, and Flash LSOs to detect and categorize trackers. It has support for crawling a list of websites to an arbitrary link depth and for performing a series of search engine keyword searches and visiting the top hit of the returned search results. We used this tool to perform a series of analyses between September and October of 2011; unless otherwise noted, our discussion reflects only behaviors observed during that time.

In presenting the results of these measurements, we make a distinction between *pages* and *domains*. Two pages may belong to the same domain (e.g., `www.cnn.com/article1` and `www.cnn.com/article2`). Which we use depends on whether we are interested in the characteristics of websites (domains) or in specific instances of tracking behavior (pages).

Note that the tracking behavior that we observe in our measurements is a lower bound, for several reasons. First, we do not log into any sites or click any ads or social widgets, which we have observed in small case studies to occasionally trigger additional tracking behavior. Second, we have observed that tracking behavior can be nondeterministic, largely due to the interplay of Behavior B (Vanilla) and Behavior D (Referred) trackers; we generally visit pages only twice (see below), which may not trigger all trackers embedded by a given website.

Finally, the mere presence of a cookie (or other storage item) does not by itself give a tracker the ability to create a browsing profile — the storage item must contain a unique identifier. It is difficult or impossible to identify unique identifiers with complete certainty (we do not reverse-engineer cookie strings), but we identify and remove any suspected trackers whose cookies or other storage contain identical values across multiple measurements that started with a clean browser. We also remove trackers that only use session cookies, though we note that these can equally be used for tracking as long as the browser remains open.

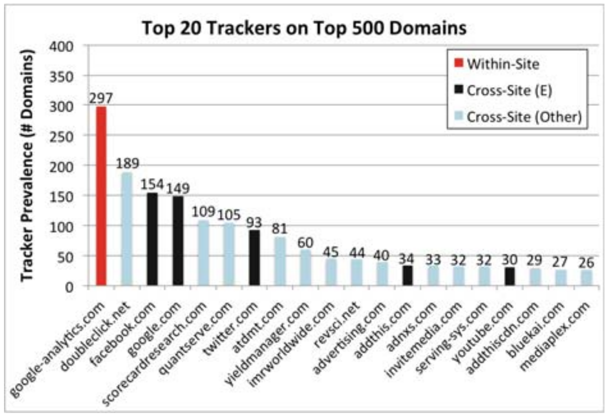


Figure 6: *Prevalence of Trackers on Top 500 Domains.* Trackers are counted on domains, i.e., if a particular tracker appears on two pages of a domain, it is counted once.

4.1 Tracking on Popular Sites

We collected a data set using the top 500 websites (international) from Alexa as published on September 19, 2011. We also visited four random links on each of the 500 sites that stayed within that site's domain. We visited and analyzed a total of 2098 unique pages for this data set; we did not visit a full 2500 unique pages because some websites do not have four within-domain links, some links are broken or redirect to other domains or to the same page, etc. This process was repeated twice: once starting with a clean browser, and once more after priming the cache and cookie database (i.e., without first clearing browser state). This experimental design aims to ensure that trackers that may only set but not read state the first time they are encountered are properly accounted for by TrackingTracker on the second run. The results we report include tracking behavior measured only on the second run.

Most of the 2098 pages (500 domains) embed trackers, often several. Indeed, the average number of trackers on the 1655 pages (457 domains) that embed at least one tracker is over 4.5 (over 7). Of these, 1469 pages (445 domains) include at least one cross-site tracker.

Overall, we found a total of 524 unique trackers appearing a cumulative 7264 times. Figure 6 shows the twenty top trackers across the 500 top domains. This graph considers websites as domains — that is, if a given tracker was encountered on two pages of a domain, it is only counted once in this graph. The most prevalent tracker is Google Analytics, appearing on almost 300 of the 500 domains — recall that it is a within-site tracker, meaning that it cannot link users' visits across these pages using cookies. The most popular cross-site tracker that users don't otherwise visit directly is Doubleclick (also owned by Google), which can track users across almost 40% of the 500 most popular sites. The most popular Behavior E tracker (domains that are themselves in the top 500) is Facebook, followed closely by Google, both of which are found on almost 30% of the top sites.

	Top 500 Sites		Non-Top 500 Sites		Popups Blocked		Cookies Blocked		No JavaScript		DNT Enabled	
Tracker Type	*#*	*Instances (Min, Max)*	*#*	*Instances (Min, Max)*	*#*	*Instances (Min, Max)*	*#*	*Instances (Min, Max)*	*#*	*Instances (Min, Max)*	*#*	*Instances (Min, Max)*
A	17	49 (1, 9)	10	34 (1, 18)	17	34 (1, 10)	40	158 (1, 38)	–	–	10	39 (1, 9)
A B	18	152 (1, 21)	11	104 (1, 37)	20	338 (1, 123)	–	–	–	–	14	105 (1, 17)
A B D	1	317 (317, 317)	1	155 (155, 155)	1	319 (319, 319)	–	–	–	–	1	274 (274, 274)
A E	8	47 (1, 17)	2	25 (5, 20)	8	51 (1, 20)	10*	95 (1, 23)	–	–	6	33 (1, 17)
A E D	1	21 (21, 21)	–	–	1	19 (19, 19)	–	–	–	–	1	18 (18, 18)
A D	3	902 (1, 896)	2	908 (55, 853)	2	906 (10, 896)	1	844 (844, 844)	–	–	2	900 (81, 819)
B	357	3322 (1, 375)	299	3734 (1, 777)	336	2859 (1, 382)	1	15 (15, 15)	161	1697 (1, 263)	320	2613 (1, 305)
B C	3	79 (6, 64)	–	–	3	29 (3, 22)	5*	48 (2, 21)	–	–	6	47 (2, 15)
B D	8	703 (1, 489)	7	60 (1, 25)	22	1235 (1, 494)	–	–	2	23 (10,13)	13	1299 (3, 551)
E	101	1564 (1, 397)	41	1569 (1, 446)	101	1625 (1, 405)	96*	1509 (1, 383)	49	707 (1, 195)	100	1412 (1, 338)
E C	1	34 (34, 34)	1	23 (23, 23)	–	–	–	–	–	–	1	31 (31, 31)
E D	1	1 (1, 1)	1	417 (417, 417)	1	5 (5, 5)	1*	1 (1, 1)	–	–	1	4 (4, 4)
C	4	4 (1, 1)	4	4 (1, 1)	–	–	5	8 (1, 4)	–	–	7	13 (1, 4)
D	1	(69, 69)	3	60 (1, 57)	2	80 (1, 79)	1*	71 (71, 71)	–	–	1	42 (42, 42)
Total	524	7264 (1, 896)	382	7093 (1, 853)	514	7505 (1, 896)	160	2749 (1, 844)	212	2427	483	6830 (1, 819)

Table 3: *Measurement Results.* Each set of columns reports the results for the specified measurement, each run with the Alexa top 500 sites except the Non-Top dataset. The lefthand column for each set reports the number of unique trackers of each type observed; the righthand column reports the number of occurrences of that tracker type. A value of `X (Y, Z)` in that column means that `X` occurrences of trackers of this type were observed; the minimum number of occurrences of any unique tracker was `Y` and the maximum `Z`. Values with asterisks would be zero (or shifted to another type) for Firefox users, due to that browser's stricter third-party cookie blocking policy. Some of the variation in counts across runs is due to the nondeterminism of tracking behavior.

Recall that a tracker may exhibit different behavior across different occurrences, often due to varying business and embedding relationships. We thus consider occurrences in addition to unique trackers; this data is reported in the first set of columns in Table 3. In this table, a tracker classified as, for instance, type A B D, may exhibit different combinations of the three behaviors at different times.

We find that most trackers behave uniformly across occurrences. For example, the most common tracking behaviors are Behavior B (Vanilla) and Behavior E (Personal), which these trackers exhibit uniformly. Some trackers, however, exhibit nonuniform behavior. For instance, sites may choose whether or not to use Quantserve for on-site analytics (Behavior A) in addition to including it as a Behavior B (cross-site) tracker. Thus, Quantserve may sometimes appear as Behavior A, sometimes as Behavior B, and sometimes as both. When other trackers include Quantserve, it can also exhibit Behavior D (Referred) behavior. Similarly, Google Analytics generally exhibits Behavior A behavior, setting site-owned state on a site for which it provides third-party analytics. However, when a third-party tracker uses Google Analytics itself, as described in Section 3.2, Google Analytics is put into the position of a Behavior D (cross-site) tracker.

4.1.1 Other Storage Mechanisms

LocalStorage. Contrary to expectations that trackers are moving away from cookies to other storage mechanisms, we found remarkably little use of HTML5 LocalStorage by trackers (though sites themselves may still use it for self-administered analytics). Of the 524 unique trackers we encountered on the Alexa top 500 sites, only eight of these trackers set any LocalStorage at all. Five contained unique identifiers, two contained timestamps, and one stored a user's unsubmitted comments in case of accidental navigation away from the page.

We observed both Behavior A and Behavior B behavior using LocalStorage. Notably, we discovered that `taboolasyndication.com` and `krxd.net` set site-owned LocalStorage instead of browser cookies for Behavior A purposes.

Of the five trackers that set unique identifiers in LocalStorage, all duplicated these values in cookies. When the same identifier is stored in multiple locations, the possibility of *respawning* is raised: if one storage location is cleared, the tracker can repopulate it with the same value from the other storage location. Respawning has been observed several times in the wild [2, 21] as a way to subvert a user's intention not to be tracked and is exemplified by the proof-of-concept evercookie[3].

We manually checked for respawning behavior in these five cases and found that one tracker — `tanx.com` — indeed repopulated the browser cookies from LocalStorage when cleared. We also noticed that `twitter.com`, which set a uniquely identifying "guest id" on the machines of users that are not logged in, did not repopulate the cookie value — however, it did store in LocalStorage the entire history of guest ids, allowing the user's new guest id to be linked to the old one. This may not be intentional on Twitter's part, but the capability for tracking in this way — equivalent to respawning — exists.

We also observed reverse respawning, that is, repopulating LocalStorage from cookies when cleared. We observed this in three of the five cases, and note that it may not be intentional respawning but rather a function of when LocalStorage is populated (generally after checking if a cookie is set and/or setting a cookie).

Flash Storage. Flash LSOs, or "cookies", on the other hand, are more commonly used to store unique tracking identifiers. Nevertheless, despite media buzz about iden-

[3]http://samy.pl/evercookie/

tifier respawning from Flash cookies, we find that most unique identifiers in Flash cookies do not serve as backups to traditional cookies; only nine of the 35 trackers with unique identifiers in Flash cookies duplicate these identifiers across Flash cookies and traditional cookies.

For these nine trackers, we tested manually for respawning behavior as described above. We observed Flash-to-cookie respawning in six cases and cookie-to-Flash respawning in seven.

In one interesting case, we found that while the Flash cookie for `sodahead.com` does not appear to match the browser cookie, it is named `enc_data` and may be an encrypted version of the cookie value. Indeed, `sodahead.com` respawned the browser cookie from the Flash cookie. Furthermore, the respawned cookie was a session cookie that would ordinarily expire automatically when the browser is closed. This example demonstrates that it is not sufficient to inspect stored values but that respawning must be determined behaviorally.

4.1.2 Cookie Leaks, Countries, and More

Throughout our study, we made a number of interesting non-quantitative observations; we describe these here.

Frequent cookie leaks. We observed a large number of cookie leaks, i.e., cookies belonging to one domain that are included in the parameters of a request to another domain, thereby circumventing the same-origin policy. Fundamentally, cookie leaking enables an additional party to gain tracking capabilities that it would not otherwise have. In addition to Behavior A leaks (the leaking of site-owned state set by the tracker's code to the tracker as a third-party) and Behavior D leaks (to enable additional trackers), we observed cookie leaks indicative of business relationships between two (or few) parties.

For example, `msn.com` and `bing.com`, both owned by Microsoft, use cookie leaking mechanisms within the browser to share cookies with each other, even when the user does not visit both sites as part of a contiguous browsing session. This enables Microsoft to track a unique user across both MSN and Bing, as well as across any site that may embed one of the two.

As another example, we noticed that when a website includes both Google AdSense (a product that allows the average website owner to embed ads without a full-fledged Doubleclick contract) as well as Google Analytics, the AdSense script makes requests to Doubleclick to fetch ads. These requests include uniquely identifying values from the site's Google Analytics cookies. This practice gives Google the capability to directly link the unique identifier used by Doubleclick to track the user across sites with the unique Google Analytics identifier used to track the user's visits to this particular site. While this does not increase the size of the browsing profile that Doubleclick can create, it allows Google to relink the two profiles if the user ever clears client-side state for one but not the other.

Origin countries. In exploring the use of LocalStorage and Flash cookies, we found that trackers from different regions appear to exhibit different behaviors. The only tracker to respawn cookies from LocalStorage comes from a Chinese domain, and of the eight trackers involved in respawning to or from Flash cookies, four are US, two are Chinese, and two are Russian. The Chinese and Russian trackers seem to be overrepresented compared to their fraction in the complete set of observed trackers.

Tracker clustering. While many of the top trackers are found across sites of a variety of categories and origins, we observed some trackers to cluster around related sites. For example, in the Alexa top 500, `traffichaus.com` and `exoclick.com` are found only on adult sites (on five and six of about twenty, respectively). Similarly, some trackers are only found on sites of the same geographic origin — e.g., `adriver.ru` is found only on Russian sites and `wrating.com` only on Chinese sites.

Trackers interact with two types of users. We observed that Behavior B (Vanilla) and Behavior C (Forced) trackers sometimes do not set tracking cookies when their websites are visited directly — unlike Behavior E tracker like Facebook, which by definition set state when they are visited. In other words, for example, `turn.com` sets a third-party tracking cookie when it is embedded on another website, but not when the user visits `turn.com` directly. Some trackers, in fact, use different domains for their own homepages than for their tracking domains (e.g., visiting `doubleclick.net` redirects to `google.com/doubleclick`). Trackers that exhibit these behaviors can never be Behavior E trackers, even if the users directly visits their sites. We can only speculate about the reasons for these behaviors, but we observe that trackers interact with two types of users: users whom they track from a third-party position, and users who are their customers and who visit their website directly.

4.2 Comparison to Less Popular Sites

The measurements presented thus far give us an intuition about the prevalence and behavior of trackers on popular sites. Since it is possible that different trackers with different behavior are found on less popular sites, we collected data for non-top sites as well. In particular, we visited 500 sites from the Alexa top million sites, starting with site #501 and at intervals of 100. As in the top 500 case, we visited 4 random links on each page, resulting in a total of 1959 unique pages visited.

In this measurement, we observed 7093 instances of tracking across 382 unique trackers, summarized in the second set of columns in Table 3. Figure 7 shows the top 20 trackers (counted by domains) for this measurement.

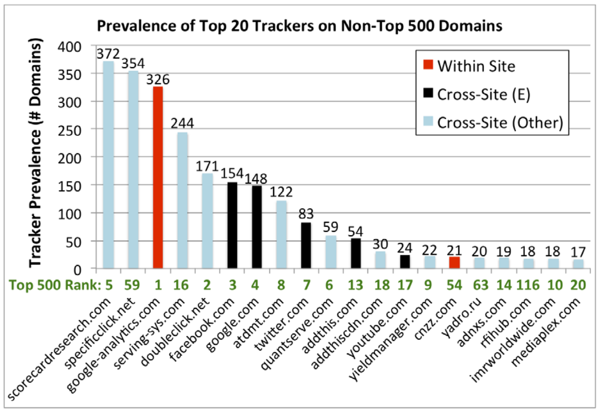


Figure 7: *Tracker Prevalence on Non-Top 500 Domains.* Trackers are counted on domains, i.e., if a particular tracker appears on two pages of a domain, it is counted once.

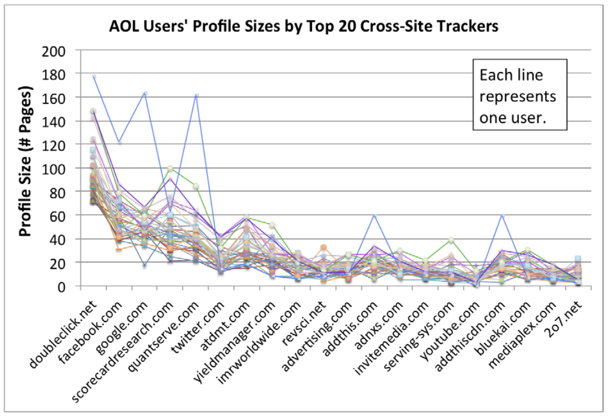


Figure 8: *Browsing Profiles for 35 AOL Users.* We report the measured profile size for each user for the 20 top trackers from the top 500, using 300 unique queries (an average of 253 unique pages visited) per user.

The numbers below each bar indicate the rank for the tracker in the top 500 domains. Note that Google Analytics and Doubleclick are no longer ranked as high, but in absolute numbers appear a similar number of times. ScorecardResearch and SpecificClick appear to be highly prevalent among among these less popular sites.

Among the non-top 500 sites, we observed less LocalStorage use — only one of the eight users of LocalStorage in the top 500 sites reappeared (`disqus.com`, which stored comment drafts); we saw one additional instance of LocalStorage, `contextweb.com`, which stored a unique value but does not duplicate it in the browser cookie. We also observed fewer Flash cookies set (68 total across all sites and trackers, compared to 110 in the top 500 measurement), finding one additional tracker — `heias.com` — that respawns its browser cookie value from the Flash cookie.

4.3 Real Users: Using the AOL Search Logs

In order to better approximate a real user's browsing history, we collected data using the 2006 AOL search query logs [20]. We selected 35 random users (about 1%) from the 3447 users with at least 300 unique queries (not necessarily clickthroughs). For each of these randomly selected users, we submitted to a search engine the first 300 of their unique queries and visited the top search result for each. This resulted in an average of 253 unique pages per user.

For the AOL users, we are interested in the size of the browsing profiles that trackers can create. Here we must consider exactly how we define "profile". In particular, a tracker receives information about the domains a user visits, the pages a user visits, and the individual visits a user makes (i.e., returning to the same page at a later time). A user may be concerned about the privacy of any of these sets of information; in the context of this study, we consider unique pages. That is, we consider the size of a browsing profile compiled by a given tracker to be the number of unique pages on which the user encountered that tracker. The reason for using pages instead of visits is that using search logs to approximate real browsing behavior involves making multiple visits to pages that a real user might not make — e.g., multiple unique queries may result in the same top search hit, which TrackingTracker will visit but a real user may not, depending on why a similar query was repeated. Though TrackingTracker may thus visit multiple pages more than once, giving more trackers the opportunity to load on that page, this is balanced by the fact that we, as before, visit each page once before recording measurements in order to prime the cache and the cookie database.

In order to compare AOL users, we focus on the top 20 cross-site trackers from the Alexa top 500 measurement. That is, we take all 19 cross-site trackers from Figure 6 as well as `serving-sys.com`, the next-highest ranked cross-site tracker. Figure 8 shows the size of the profile compiled by each tracker for each of the 35 users.

We find that Doubleclick can track a user across (on average) 39% of the pages he or she visited in these browsing traces — and a maximum of 66% of the pages visited by one user. The magnitude of these percentages may be cause for concern by privacy-conscious users. Facebook and Google can track users across an average of 23% and 21% of these browsing traces (45% and 61% in the maximum case), respectively. As many users have and are logged into Facebook and Google accounts, this tracking is likely not to be anonymous.

Two data points of note are the large profile sizes for `google.com` and `quantserve.com` for one of the users. These spikes occur because that user visited a large number of pages on the same domain (`city-data.com`), which embeds Google Maps and Quantserve.

From this data, we observe that, in general, the ranking of the trackers in the top 500 corresponds with how much real users may encounter them. In particular, `doubleclick.net` remains the top cross-site tracker; the prominence of `scorecardresearch.com` in the non-top 500 is not reflected here, perhaps because top search hits are likely biased towards more popular sites.

5 Defenses

In this section, we explore existing defenses against tracking in the context of our classification. We then present measurement results collected using the Alexa top 500 with standard defenses enabled. Finally, we propose an additional defense — implemented in the form of our Firefox add-on ShareMeNot — that aims to protect users from Behavior E tracking. Again, unless otherwise noted, we refer to observations in September/October 2011.

5.1 Initial Analysis of Defenses

Third-party cookie blocking. A standard defense against third-party web tracking is to block third-party cookies. This defense is insufficient for a number of reasons. First, different browsers implement third-party cookie blocking with different degrees of strictness. While Firefox blocks third-party cookies both from being *set* as well as from being *sent*, most other browsers (including Chrome, Safari, and Internet Explorer) only block the setting of third-party cookies. So, for example, Facebook can set a first-party cookie when the user visits `facebook.com`; in browsers other than Firefox, this cookie, once set, is available to Facebook from a third-party position (when embedded on another page).

Thus, in most browsers, third-party cookie blocking protects users only from trackers that are never visited directly — that is, it is effective against Behavior B (Vanilla) trackers but not against Behavior C (Forced) or Behavior E (Personal) trackers. Firefox's strict policy provides better protection, but at the expense of functionality like social widgets and buttons, some instantiations of OAuth or federated login, and other legitimate cross-domain behavior (thus prompting Mozilla to opt against making this setting the default [19]).

Do Not Track. The recently proposed Do Not Track header and legislation aim to give users a standardized way to opt out of web tracking. A browser setting (already implemented natively in Firefox, IE, and Safari) appends a DNT=1 header to outgoing requests, informing the receiving website that the user wishes to opt out of tracking. As of February 2012, Do Not Track is merely a policy technique that requires tracker compliance, providing no technical backing or enforcement. A major sticking point is the debate over the definition of tracking, as the conclusion of this debate determines to which parties the Do Not Track legislation will apply. As evidenced by the papers submitted to the W3C Workshop on Web Tracking and User Privacy[4], many of the parties involved in tracking argue that their behaviors should not be considered tracking for the purposes of DNT. It is our hope that the tracking classification framework that we have developed and proposed in this paper can be used to further the discussion of what should be considered tracking in the policy realm, and that a tool like TrackingTracker can be used in the browser to enforce and detect violations of Do Not Track.

Clearing client-side state. There has been some concern [26] that pervasive opt-out of tracking will create a tiered or divided web, in which visitors who opt out of tracking (via the DNT header or other methods) will not be provided with the same content as other visitors. One possible solution (also identified in [9]) to avoid tracking in the face of this concern is to constantly clear the browser's client-side state, regularly receiving new identifiers from trackers. This may be a sufficient solution for Behavior B, Behavior C, and Behavior D trackers, but it cannot protect users against Behavior E trackers to which they have identified themselves as a particular account holder (and thus logging back in will re-identify the same user). It is also hard to implement against Behavior A trackers, as they set first-party state on the websites that embed them, and it is difficult to distinguish in a robust manner the first-party state needed by the website from the state used by the Behavior A tracker. Other work [25] shows furthermore that fingerprinting techniques can re-identify a large fraction of hosts with fresh cookies.

Blocking popups. Most browsers today block popups by default, potentially making it more difficult for Behavior C trackers to maneuver themselves into first-party positions. However, websites can still open popups in response to user clicks. Furthermore, popups are only one way that Behavior C trackers can force a user to visit their site directly (and the easiest of these to detect and block). Other methods include redirecting the user's browser to the tracker's domain and back using javascript, or business relationships between the tracker and the embedding site that involve the site redirecting directly to a full-page interstitial ad controlled by the tracker's domain. These behaviors are hard or impossible to block as they are used throughout the web for other legitimate purposes.

Recent findings (February 2012) [18] furthermore revealed programmatic form submission as a new technique for Behavior C tracking in Safari, which treats form submissions as first-party interactions.

Private browsing mode. Private browsing mode, as explored in depth in [1], does not primarily address the threat model of web tracking. Instead, private browsing mode aims to protect browser state from adversaries with physical access to the machine. While the clearing of cookies when exiting private browsing mode can help increase a user's privacy in the face of tracking, private browsing mode does not aim to keep a user's browsing history private from remote servers.

[4]`http://www.w3.org/2011/track-privacy/`

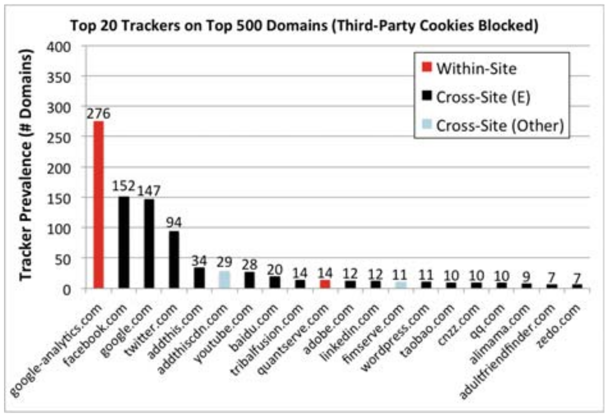


Figure 9: *Prevalence of Trackers on Top 500 Domains with Third-Party Cookies Blocked.* Trackers are counted on domains, that is, if a particular tracker appears on two pages of a domain, it is counted once.

5.2 Empirical Analysis of Defenses

As a part of our measurement study, we empirically analyzed the effectiveness of popup blocking and third-party cookie blocking to prevent tracking. The results of these measurements (run using the Alexa Top 500 domains, with 4 random links chosen from each) are summarized on the righthand side of Table 3. Overall, we find that existing defenses protect against a large portion of tracking, with the notable exception of Behavior E. We dive into the effects of each measured defense in turn.

With popups blocked, we did not observe significant differences in the tracking capabilities of most trackers. As expected, we observe fewer trackers exhibiting Behavior C (Forced) — however, Behavior C using redirects remains, leaving three trackers exhibiting such behavior. We find most Behavior C trackers exhibit this type of behavior only occasionally, acting as Behavior B (Vanilla) the rest of the time. Thus, with third-party cookies enabled, popup blocking does not affect the capabilities of most trackers. Indeed, we suspect that most popups are used to better capture the user's attention rather than to maneuver the tracker domain into a first-party position. Nevertheless, this technique is sometimes used for this purpose[5].

Third-party cookie blocking is, as expected, a better defense against tracking. However, recall that in most browsers other than Firefox, third-party cookie blocking only blocks the setting, not the sending, of cookies. Thus, if a tracker can ever set a cookie (via Behavior C or Behavior E), this cookie is available from that point forward. In Table 3, we distinguish the results for Firefox's strict cookie blocking policy: any type with an asterisk in the "Cookies Blocked" column disappears in Firefox (trackers that exhibit both Behavior A and E reduce to only A; the others disappear). Note the presence of one Behavior B tracker: this is `meebo.com`,

[5]`http://stackoverflow.com/questions/465662/`

Figure 10: *Example Social Widgets.* Behavior E trackers expose social widgets that can be used to track users across all the sites on which these widgets are embedded.

which sets unique identifiers in LocalStorage in addition to browser cookies, leaving it unaffected by cookie blocking. Figure 9 shows the top 20 trackers for this measurement (compare to Figure 6), in which it is evident that most cross-site trackers have disappeared from the top 20, leaving the prominence of Behavior E trackers like Facebook, Twitter, YouTube, and others.

We also measured the effectiveness of disabling JavaScript, the most blunt defense against tracking. We find that it is extremely effective at preventing tracking behaviors that require API access to cookies to leak them, as is the case for Behavior A (Analytics) and Behavior D (Referred). However, trackers can still set cookies via HTTP headers and Behavior C trackers can use HTML redirects. Any tracking that requires only that content be requested from a tracker is not impacted — thus, while the scripts of Behavior E and other trackers cannot run (e.g., to render a social widget), they can be requested, thereby enabling tracking. Some trackers simply use `<noscript>` tags to fetch single-pixel images ("beacons") when more complex scripting techniques are not available. Despite being the most effective single defense, disabling JavaScript renders much of today's web unusable, making it an unworkable option for most users.

The Do Not Track header does not yet appear to have a significant effect on tracking, as evidenced by the sustained prevalence of most trackers in Table 3. Note that, as in all the results we report, we did exclude any trackers that set only non-unique and/or session cookies, as some trackers may respond to the DNT header by setting an opt-out cookie. We did notice that a few fairly prevalent trackers appeared to respond to the header, including `gemius.pl`, `serving-sys.com`, `media6degrees.com` and `bluekai.com`. These results are consistent with a recent set of case studies of DNT compliance [17].

Finally, we note that we did not observe any trackers actively changing behavior in an attempt to circumvent the tested defenses — that is, we did not observe more LocalStorage or more Flash cookies. Though we have not verified whether or not these trackers instead use more exotic storage mechanisms like cache Etags, we hypothesize that enough users do not enable these defenses to mobilize trackers to substantially change their behavior, and hence fall outside our common case explorations.

5.3 A New Defense: ShareMeNot

From these measurements, we conclude that a combination of defenses can be employed to protect against a large set of trackers. However, Behavior E trackers like

Tracker	*Without ShareMeNot*	*With ShareMeNot*
Facebook	154	9
Google	149	15
Twitter	93	0
AddThis	34	0
YouTube	30	0
LinkedIn	22	0
Digg	8	0
Stumbleupon	6	0

Table 4: *Effectiveness of ShareMeNot.* ShareMeNot drastically reduces the occurrences of tracking behavior by the supported set of Behavior E trackers.

Facebook, Google, and Twitter remain largely unaffected. Recall that these trackers can track logged-in users non-anonymously across any sites that embed so-called "social widgets" exposed by the tracker. For example, Facebook allows other websites to embed a "Like" button, Google exposes a "+1" button, and so on (see Figure 10 for a number of examples). These buttons present a privacy risk for users because they track users even when they choose not to click on any of the buttons.

When users can protect themselves from tracking in this fashion, it comes at the expense of the functionality of the button. In Firefox, the stricter third-party cookie blocking policy renders the buttons unusable. Other existing defenses, including the popular Disconnect browser extension[6], work by simply blocking the tracker's scripts and their associated buttons from being loaded by the browser at all, thereby effectively removing the buttons from the user's web experience entirely.

We introduce ShareMeNot, a Firefox add-on that aims to find a middle ground between allowing the buttons to track users wherever they appear and retaining the functionality of the buttons when users explicitly choose to interact with them. It does this by stripping cookies from third-party requests to any of the supported Behavior E trackers under ordinary circumstances (as well as from any other blacklisted requests that are made in the context of loading such a button); when it detects that a user has clicked on a button, it allows the cookies to be included with the request, thereby allowing the button click to function as normal, transparently to the user.

The use of ShareMeNot shrinks the profile that the supported Behavior E trackers can create to only those sites on which the user explicitly clicks on one of the buttons — at which point the button provider must necessarily know the user's identity and the identity of the site on which the button was found in order to link the "like" or the "+1" action to the user's profile. No other existing approach can both shrink the profile a Behavior E tracker can create while also retaining the functionality of the buttons, though concurrent work on the Priv3 Firefox add-on [3] adopts the same basic approach; as of February 2012, Priv3 supports fewer widgets and, to our knowledge, was not iteratively refined through measurement.

We experimentally verified the effectiveness of ShareMeNot. As summarized in Table 4, ShareMeNot dramatically reduces the presence of the Behavior E trackers it supports to date. We chose to support these sites based in part on our initial, pre-experimental perceptions of popular third-party trackers, and in part based on our experimental discovery of the top trackers. ShareMeNot entirely eliminates tracking by most of these, including Twitter, AddThis, Youtube, Digg, and Stumbleupon. While it does not entirely remove the presence of Facebook and Google, it reduces their prevalence to 9 and 15 occurrences, respectively. In the Facebook case, this is due to the Facebook comments widget, which triggers additional first-party requests (containing tracking information) not blacklisted by ShareMeNot; the Google cases appear mostly on other Google domains (e.g., `google.ca`).

The currently released ShareMeNot add-on does not fully block requests to the trackers, thus exposing the user's IP address and other fingerprinting information, nor does it block programmatic access to `document.cookie`. A new version of ShareMeNot is under development that aims to address these issues by replacing widgets entirely with client-side buttons, making no requests to trackers until these replacement buttons are clicked.

As of February 2012, we have seen over 20,000 downloads from our own servers[7], in addition to over 7000 daily users as reported by the official Mozilla add-on site[8].

6 Related Work

We expand our discussions of several related works and consider additional related works not discussed above.

A number of studies have empirically examined tracking on the web, most notably [14]. In that paper, the authors present the results of a longitudinal measurement study of web tracking, examining the prevalence of third-party trackers on the web. The authors do not distinguish between different types of trackers, grouping together, for example, Google Analytics (a within-site tracker) and Doubleclick (a cross-site tracker), though they touch on aspects in prior work [15]. As discussed, we believe that this distinction is fundamentally important for understanding and responding to web tracking.

In their five-year study of modern web traffic, Ihm and Pai [8] find that ad network traffic accounts for a growing percentage of total requests (12% in 2010). They find Google Analytics on up to 40% of the pages reflected in their data, a number that has increased to over 50% in our data. Another measurement study of web tracking appeared in [12], in which the authors examined the prevalence of cookie usage and P3P policies.

[6]`http://disconnect.me`

[7]`http://sharemenot.cs.washington.edu/`

[8]`https://addons.mozilla.org/firefox/addon/sharemenot/`

From a slightly different threat model, the authors of [11] examined privacy-violating information flows on the web, though they don't distinguish third-party trackers from visited sites themselves. As in our study, they found a number of instances of cookie leaking, as well as on-site behavioral tracking and other privacy violations. In [13] and [16], the authors examine the direct leakage of private data from first-party websites to data aggregators, including the potential linkage of user accounts on separate sites.

In [9], the authors classify trackers based on cooperation between the embedding site and the trackers, which in some ways overlaps with our classification. They do not measure the prevalence of these tracker classes, and miss Behavior E (Personal), which has only emerged in popularity since the publication of that paper.

Further afield, a number of researchers [5, 7, 23] have tackled the problem of privacy-preserving targeted advertising and other personalized content, attempting to find a middle ground that balances the values of users, websites, and advertisers or other content providers.

Additionally, there have been significant online discussions about tracking, e.g., [17]. Finally, entire workshops on tracking have emerged, e.g., the 2011 Workshop on Internet Tracking, Advertising, and Privacy and the 2011 W3C Workshop on Web Tracking and User Privacy.

7 Conclusion

In this paper we presented an in-depth empirical investigation of third-party web tracking. Our empirical investigation builds on the introduction of what we believe to be the first comprehensive classification framework for web tracking based on client-side observable behaviors. We believe that this framework can serve as a foundation for future technical and policy initiatives. We additionally evaluated a set of common defenses on a large scale and observed a gap — the ability to defend against Behavior E tracking with social media widgets, like the Facebook "Like" button, while still allowing those widgets to be useful. In response, we developed and evaluated ShareMeNot, which is designed to thwart such tracking while still allowing the widgets to be used.

Acknowledgements

We thank our shepherd, Jon Crowcroft, and the anonymous NSDI reviewers for their feedback. We thank Daniel Halperin, Arvind Narayanan, and Charlie Reis for feedback on earlier drafts; Chris Rovillos for volunteering to help maintain and extend ShareMeNot; and Brandon Lucia for naming it. This work was supported in part by NSF Awards CNS-0722000, CNS-0846065, and CNS-0917341, an NSF Graduate Research Fellowship under Grant No. DGE-0718124, an Alfred P. Sloan Research Fellowship, and a gift from Google.

References

[1] G. Aggrawal, E. Bursztein, C. Jackson, and D. Boneh. An analysis of private browsing modes in modern browsers. In *Usenix Security Symposium*, 2010.

[2] M. Ayenson, D. J. Wambach, A. Soltani, N. Good, and C. J. Hoffnagle. Flash Cookies and Privacy II: Now with HTML5 and ETag Respawning. *Social Science Research Network Working Paper Series*, 2011.

[3] M. Dhawan, C. Kreibich, and N. Weaver. The Priv3 Firefox Extension. http://priv3.icsi.berkeley.edu/.

[4] P. Eckersley. How unique is your web browser? In *International Conference on Privacy Enhancing Technologies*, 2010.

[5] M. Fredrikson and B. Livshits. RePriv: Re-Envisioning In-Browser Privacy. In *IEEE Symp. on Security and Privacy*, 2011.

[6] A. Goldfarb and C. E. Tucker. Privacy Regulation and Online Advertising. *Management Science*, 57(1), Jan. 2011.

[7] S. Guha, B. Cheng, and P. Francis. Privad: Practical Privacy in Online Advertising. In *NSDI*, 2011.

[8] S. Ihm and V. Pai. Towards understanding modern web trafÞc. In *IMC*, 2011.

[9] C. Jackson, A. Bortz, D. Boneh, and J. C. Mitchell. Protecting browser state from web privacy attacks. In *WWW*, 2006.

[10] A. Janc and L. Olejnik. Feasibility and real-world implications of web browser history detection. In *W2SP*, 2010.

[11] D. Jang, R. Jhala, S. Lerner, and H. Shacham. An empirical study of privacy-violating information flows in JavaScript web applications. In *CCS*, 2010.

[12] C. Jensen, C. Sarkar, C. Jensen, and C. Potts. Tracking website data-collection and privacy practices with the iWatch web crawler. In *SOUPS*, 2007.

[13] B. Krishnamurthy and C. Wills. On the leakage of personally identifiable information via online social networks. In *WOSN*, 2009.

[14] B. Krishnamurthy and C. Wills. Privacy diffusion on the web: a longitudinal perspective. In *WWW*, 2009.

[15] B. Krishnamurthy and C. E. Wills. Generating a privacy footprint on the internet. In *IMC*, 2006.

[16] B. Krishnamurthy, K. Naryshkin, and C. Wills. Privacy leakage vs. protection measures: the growing disconnect. In *W2SP*, 2011.

[17] J. Mayer. Tracking the Trackers: Early Results, 2011. http://cyberlaw.stanford.edu/node/6694.

[18] J. Mayer. Safari tracking, Jan. 2012. http://webpolicy.org/2012/02/17/safari-trackers/.

[19] Mozilla. Bug 417800 — Revert to not blocking third-party cookies, 2008. https://bugzilla.mozilla.org/show_bug.cgi?id=417800.

[20] G. Pass, A. Chowdhury, and C. Torgeson. A Picture of Search. In *Conf. on Scalable Information Systems*, 2006.

[21] A. Soltani, S. Canty, Q. Mayo, L. Thomas, and C. J. Hoofnagle. Flash Cookies and Privacy. *Social Science Research Network Working Paper Series*, Aug. 2009.

[22] ThreatMetrix. Tech. overview. http://threatmetrix.com/technology/technology-overview/.

[23] V. Toubiana, A. Narayanan, D. Boneh, H. Nissenbaum, and S. Barocas. Adnostic: Privacy Preserving Targeted Advertising. In *NDSS*, 2010.

[24] R. Vamosi. Device Fingerprinting Aims To Stop Online Fraud. PCWorld, Mar. 2009. http://www.pcworld.com/businesscenter/article/161036/.

[25] T.-F. Yen, Y. Xie, F. Yu, R. P. Yu, and M. Abadi. Host fingerprinting and tracking on the web: Privacy and security implications. In *NDSS*, 2012.

[26] H. Yu. Do Not Track: Not as Simple as it Sounds, Aug. 2010. https://freedom-to-tinker.com/blog/harlanyu/do-not-track-not-simple-it-sounds.

Towards Statistical Queries over Distributed Private User Data

Ruichuan Chen[†] Alexey Reznichenko[†] Paul Francis[†] Johannes Gehrke[§]
[†]*Max Planck Institute for Software Systems (MPI-SWS), Germany*
[§]*Cornell University, Ithaca, NY 14853, USA*
{rchen, areznich, francis}@mpi-sws.org, johannes@cs.cornell.edu

Abstract

To maintain the privacy of individual users' personal data, a growing number of researchers propose storing user data in client computers or personal data stores in the cloud, and allowing users to tightly control the release of that data. While this allows specific applications to use certain approved user data, it precludes broad statistical analysis of user data. Distributed differential privacy is one approach to enabling this analysis, but previous proposals are not practical in that they scale poorly, or that they require trusted clients. This paper proposes a design that overcomes these limitations. It places tight bounds on the extent to which malicious clients can distort answers, scales well, and tolerates churn among clients. This paper presents a detailed design and analysis, and gives performance results of a complete implementation based on the deployment of over 600 clients.

1 Introduction

User privacy on the Internet has become a major concern. User data is exposed to organizations in a bewildering and growing variety of ways. Sometimes the exposure is known to users, as when a user makes a purchase in an online store, or updates a profile on an online social networking site. But often users are unaware that their data is being exposed, for instance by third party trackers [34] or smart phone applications [19]. In some cases, the exposure provides benefits to users, for instance in the form of personalized recommendations. In other cases, the exposure may be detrimental to users, for instance as when Google used users' Gmail contact data to seed its Buzz social network [2].

There is a growing sense that this loss of privacy has to be brought under control. ProjectVRM, for instance, states that users must have exclusive control of their own data, and must be able to share data selectively or voluntarily [4]. This "user-owned and operated" principle has already been reflected in commercial and research efforts on various types of user data, such as individuals' healthcare data [3], time-series data [45, 49], and behavioral data used for advertising [7, 31, 51]. Here, each individual user's data is stored in a client or cloud device under the user's control, and is released in a controlled, limited, or noisy fashion. It is important to be able to make statistical queries over such *distributed* user data while still preserving privacy.

One general approach to supporting privacy-preserving statistical queries is to anonymize the user data or add noise to the user data, so that individual users cannot be identified. Unfortunately, this approach heavily restricts the utility of user data [43], and often users can be de-anonymized [1, 11, 41, 46], for instance using auxiliary information.

Another general approach, which we adopt in this paper, is to add noise to the answers of queries, in such a way that the privacy of individual users is protected. An instance of this approach that is popular in the research community is *differential privacy* [13, 14, 17]. Specifically, differential privacy adds noise to the answers of queries to statistical databases so that the querying system cannot detect the presence or absence of a single user or a set of users. Differential privacy provides a provable foundation for measuring privacy loss regardless of what information an adversary may possess. In spite of the fact that an increasing number of queries can lead to increased privacy loss, we find differential privacy to be an attractive model both because it does provide measurable privacy, and because there is substantial ongoing effort in developing its uses [32, 38, 39, 40, 47] and understanding its limitations [15].

Most work in differential privacy assumes a centralized database front-ended by a trusted query module. There is, however, no centralized database existing in a distributed setting with individual users maintaining their own data. Some form of *distributed differential privacy* is therefore required. To our knowledge, there are a few prior designs for distributed differential privacy in the literature [16, 30, 45, 49], none of which appear practical in a realistic distributed environment. The first design [16] has a per-user computational load of $O(U)$, where U is the number of users, making it impractical for large systems. Following designs [45, 49] reduce the per-user computational load from $O(U)$ to $O(1)$, but this complexity assumes that key shares have been distributed among users, using an expensive secret sharing protocol. However, in reality users often exhibit churn (go on and offline), and it would be very hard to execute such a protocol among a sizable population of such users. To tolerate churn, the recent design [30] introduces two honest-but-curious servers to collaboratively compute the query result. Nevertheless, these designs [30, 45, 49] all suffer from a common attack that even a single malicious user can substantially distort the query result.

The goal of this paper is to design, build, and measure a *practical* system that provides differentially private query semantics over distributed user data. Our system, dubbed **PDDP** for Practical Distributed Differential Privacy, assumes that *clients* are user devices under users' control. Therefore, they may be malicious, and they are not always online. An *analyst*, who wishes to make statistical queries over some number of clients, formulates a query and transmits it to an honest-but-curious *proxy*[1], which further forwards it to the specified number of clients. Each client locally executes the query, and sends its answer back to the proxy encrypted with the analyst's public key. In parallel, the proxy and clients, using an efficient XOR homomorphic bit-cryptosystem, *collaboratively* generate a set of additional noisy answers that produce the required amount of differentially private noise. The proxy then shuffles the received client answers and the indistinguishable noisy answers, and forwards them together to the analyst. Finally, the analyst decrypts them and computes the statistical result under the differentially private guarantee. Altogether, this paper makes the following contributions:

- It proposes what is to our knowledge the first distributed differentially private system, PDDP, that is practical in that it can operate at large scale with malicious clients and under churn.

- It gives a detailed privacy analysis, and presents the implementation results of a fully functional PDDP system with a realistic deployment of 600+ clients.

The rest of this paper is organized as follows. We first give the security assumptions and performance goals in §2. The system design is presented in §3. We then provide a privacy analysis of the system in §4. This is followed in §5 by the implementation description and performance evaluation based on micro-benchmarks and our 600+ client deployment. In §6, we sketch a design for how to weaken the proxy trust requirement by using trusted hardware at proxy. Finally, we give an overview of related work in §7, and outline future work in §8.

[1] We later suggest a practical approach to reduce the proxy trust (§6).

2 Assumptions and Goals

The PDDP system consists of three components: *analysts*, *clients*, and *proxy*. Analysts make queries to the system, and collect answers. Clients locally maintain their own data, and answer queries. The proxy mediates between the analysts and clients, and adds differentially private noise to clients' answers to preserve privacy.

2.1 Security Assumptions

Analysts are assumed to be potentially malicious, with a goal of violating individual users' privacy. An analyst may collude with other analysts, or pretend to be multiple distinct analysts. An analyst may take control of clients, and attempt to use the PDDP protocol to reveal information about those clients. (Of course, an analyst could reveal information about those clients outside of PDDP, but this is the case today with for instance malware and so is out of scope.) An analyst may deploy its own clients and manipulate their answers. An analyst may also publish its collected answers. Analysts can intercept and modify all messages (e.g., an ISP posing as an analyst).

Clients are also assumed to be potentially malicious, with a goal of distorting the statistical results learned by analysts. Clients may generate false or illegitimate answers under coordinated control (e.g., as a botnet), and may act as Sybils [12].

The proxy is assumed to be honest but curious (HbC). It will faithfully follow the specified protocol, but may try to exploit additional information that can be learned in so doing. The proxy does not collude with other components. We discuss how we may be able to relax the HbC assumption by using trusted hardware in §6.

It is assumed that clients have the correct public keys for analysts and the proxy, that analysts and the proxy have correct public keys for each other, and that the corresponding private keys are kept secure.

Discussion. It would be ideal if our design did not require an HbC proxy. Indeed, some prior designs for distributed differential privacy do not require such a proxy [16, 45, 49]. They achieve this, however, at an unacceptable cost in a realistic distributed setting (see §1). Here we briefly discuss the viability of an HbC proxy.

We envision a scenario whereby the analysts pay the proxy to operate, as suggested in [24]. While this admittedly leads to opportunities for collusion, we point out that such relationships already exist in industry today that normally do not result in collusion. For instance, companies pay for independent audits of their financial books or independent safety testing of their products, even though this may lead to negative consequences. We therefore believe that, in practice, the HbC arrangement is reasonable for the PDDP system in most scenarios.

2.2 Client Characteristics

The client population consists of user devices, including home and mobile devices. Clients are therefore assumed to have the churn characteristics, and the CPU and bandwidth capacities of current home and mobile devices. In other words, clients may go offline or shutdown at any time, and may have limited resources for computation and data transmission.

2.3 Goals

The primary goal of our PDDP system is that the differentially private guarantee is *always* maintained for every honest client.

The second goal is that the *maximum* absolute distortion in the final statistical result tallied by an analyst is bounded by the number of malicious clients (here ignoring distortion due to the differentially private noise itself). In other words, if z clients are malicious, then the absolute error in the statistical result will be z or less.

Finally, the system should scale for queries to a very large client base (millions of clients). While client churn may result in it taking a relatively long time to produce statistical results, this should not prevent results from being produced.

3 System Design

3.1 Differential Privacy Background

In principle, differential privacy ensures that the output of a computation does not violate the privacy of individual inputs. Formally, a computation $\mathscr{F}$ gives (ε, δ)-differential privacy [16] if, for all datasets D_1 and D_2 differing on one record, and for all outputs $S \subseteq Range(\mathscr{F})$:

$$\Pr[\mathscr{F}(D_1) \in S] \leq \exp(\varepsilon) \times \Pr[\mathscr{F}(D_2) \in S] + \delta \qquad (1)$$

In other words, for any possible record, the probability that the computation $\mathscr{F}$ produces a given output is almost independent of whether or not this record is present in the dataset.

The strong guarantees of differential privacy do not come for free. Privacy is preserved by adding noise to the output of a computation. Specifically, there are two privacy parameters: ε and δ. The former parameter ε mainly controls the tradeoff between the accuracy of a computation and the strength of its privacy guarantee. Higher ε represents higher accuracy but weaker privacy guarantee, and vice versa. The latter parameter δ relaxes the strict relative shift of probability in some cases, where the expression (1) cannot be satisfied without a non-zero δ [16].

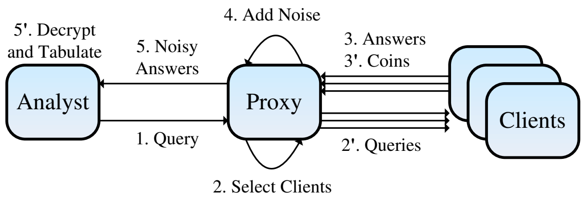


Figure 1: System components and basic protocol

Differential privacy does not make any assumptions about the adversary. It is independent of the adversary's computational power and auxiliary information. Such information has been shown to break many other privacy mechanisms [11, 41, 46].

3.2 Basic Protocol Design

3.2.1 Periodic Client-Proxy Exchange

Periodically, each online client connects to the proxy using standard session encryption (e.g., TLS) authenticated by the proxy's public key. On the first such connection, the proxy assigns a unique ID to the client, which the (honest) client uses to identify itself in subsequent connections. With each connection, the client receives and answers any pending queries from the proxy. In addition, the client sends zero or more random "tossed coins" (i.e., encrypted '0' or '1' values) to the proxy (see §3.2.3).

The proxy maintains a complete list of clients. For each client, it stores the client's ID, the client's *privacy deficit* (an indication of the client's privacy loss across all analysts, see §3.3.2), and the timestamp of when the client last connected. Clients that have not connected for a long time (weeks or months) are removed from the list.

3.2.2 Query-Answer Workflow

The query-answer workflow consists of five steps, as illustrated in Figure 1.

Step 1: Query Initialization (Analyst→Proxy). An analyst formulates a general SQL-style *query*, and transmits it to the proxy. Answers to a query will be provided as a set of b *buckets* whose lower bounds L_i and upper bounds U_i are specified by the analyst. Different buckets should not overlap. Additionally, the analyst also specifies the number of clients c that should be queried, and a privacy parameter ε:

$$A \rightarrow P : query, \{L_i, U_i\}_{i=1}^{b}, c, \varepsilon$$

Here we again assume standard session encryption to mutually authenticate proxy and analyst, and prevent man-in-the-middle attacks from distorting messages. As an illustrative example, the analyst's query "what is the distribution of ages among males?" can be formulated as an SQL-style query "`SELECT age FROM info WHERE gender='m'`" with 4 buckets, e.g., age 0$\sim$12, age 13$\sim$20, age 21$\sim$59, and age$\geq$60.

Step 2: Query Forwarding (Proxy→Client). Once the proxy receives the analyst's query, it rejects the query if c is too low or too high, or if ε exceeds the maximum allowable privacy level. Otherwise, the proxy selects c unique clients to which to send the query, under one of the following two policies (selectable by the analyst):

- Select c clients randomly from the complete list of clients, and wait for them to connect.
- Select the first c clients that connect.

Under the first policy, some of the selected clients may not connect after a long time (many hours or days). In this case, other random clients can be selected instead, until answers from c or nearly c clients are collected. Under the second policy, answers can be collected more quickly, but with a bias towards clients that are more often online. The proxy may reject client connections that occur more frequently than the defined connection period. This is to prevent malicious clients from connecting very frequently and so dominating the answer set.

After client selection, the proxy forwards the query (with bucket information) to the c selected clients when they connect to the proxy:

$$P \rightarrow C : query, \{L_i, U_i\}_{i=1}^{b}$$

Step 3: Client Response (Client→Proxy). Each client maintains its own data in a local database. Upon receiving a query, the client produces an *answer* in the form of a '1' or a '0' per bucket, depending on whether or not the answer falls within the range of that bucket. Depending on the query, more than one bucket may contain a '1'.

Each per-bucket binary value is individually encrypted using the Goldwasser-Micali (GM) bit-cryptosystem [27, 28] with the analyst's public key. The client returns this series of encrypted binary values $\{v_i\}_{i=1}^{b}$ as the actual answer to the proxy:

$$C \rightarrow P : \{v_i\}_{i=1}^{b}$$

GM is a probabilistic public-key cryptosystem. A given input produces a different ciphertext every time it is encrypted. GM has significant advantages in our system. First, it is very efficient compared with other more general public-key cryptosystems. Second, it is a single-bit cryptosystem which encodes only two possible values, 0 and 1. This enforces a binary value in each bucket, and prevents the use of individual encrypted values as a covert channel.

Upon receiving a client's answer (in the form of a series of GM-encrypted values $\{v_i\}_{i=1}^{b}$), the proxy checks the legitimacy of the answer. A legitimate GM-encrypted value must have its Jacobi symbol equal to '+1', so that the proxy can easily and efficiently detect illegitimate values [18, 48]. If a client's answer is legitimate (i.e., $\{v_i\}_{i=1}^{b}$ are all legitimate), the proxy stores the answer locally; otherwise, the proxy discards this answer.

Step 4: Differentially Private Noise Addition. To preserve clients' privacy, the proxy adds differentially private noise to each bucket. This is done by creating some number of additional binary noise-values selected from a binomial distribution with success probability $p = 0.5$. We call these additional noise-values *coins*. Enough *unbiased* coins must be added by the proxy to obtain the amount of noise required by the differential privacy. It is proven in [16] that forming a binomial distribution by adding n unbiased coins achieves (ε, δ)-differential privacy, where

$$n \geq \frac{64 \ln(\frac{2}{\delta})}{\varepsilon^2} \tag{2}$$

The value of ε is chosen by the analyst according to some accuracy/privacy trade-off (see §3.1). The value of δ may also be chosen by the analyst, but if $\delta \geq 1/c$, a single client's privacy may be compromised [33]. This may happen in the case where the query is for an attribute unique to a single client, e.g., its social security number. Applying $\delta < 1/c$ to expression (2), the number of coins for each bucket is at least:

$$n = \lfloor \frac{64 \ln(2c)}{\varepsilon^2} \rfloor + 1 \tag{3}$$

The proxy maintains a *pool* of unbiased coins (see §3.2.3). Note that all coins in this pool are GM-encrypted with the analyst's public key, and are indistinguishable from clients' answer values.

If a query has b buckets, then the proxy needs to use (and remove) in total $b \times n$ coins from the pool, i.e., to add n different coins to each bucket.

Step 5: Noisy Answers to Analyst (Proxy→Analyst). With the noise addition, in each bucket, there are $c+n$ encrypted binary values, c from clients' answers and n from added coins. After a random delay, the proxy further shuffles the $c+n$ values in each bucket *independently*. This prevents a client from being identified based on the vector of 1's and 0's in its answer, and eliminates the potential covert channel. The result is a set of shuffled encrypted values $\{e_{i,j}\}_{j=1}^{c+n}$ for each i-th bucket.

Note that, in each bucket, adding n unbiased coins introduces differentially private noise with mean equal to $n/2$. As a result, for the final aggregate answer adjustment, the proxy also informs the analyst of n, i.e., the number of coins added to each bucket. Thus, the message sent from proxy to analyst is as follows:

$$P \rightarrow A : \left\{ \left\{ e_{i,j} \right\}_{j=1}^{c+n} \right\}_{i=1}^{b}, n$$

After the proxy sends this message to the analyst, the proxy adds ε and δ to the respective privacy deficits (see §3.3.2) of each client that contributed an answer. Here, δ can be approximately considered as $1/c$ (see Step 4).

Upon receiving the message, the analyst uses its private key to decrypt all encrypted binary values, and obtains a set of plaintext values $\{d_{i,j}\}_{j=1}^{c+n}$ for each i-th bucket. Finally, for each i-th bucket, the analyst sums up all plaintext values, and then subtracts $n/2$ to get the (noisy) aggregate answer a_i:

$$a_i = \sum_{j=1}^{c+n} d_{i,j} - \frac{n}{2} \qquad (4)$$

In the end, the analyst obtains a noisy answer for how many clients fall within each bucket (i.e., a_i for the i-th bucket) under the guarantee of differential privacy. Ultimately, our design transforms any query into a counting query. Though this kind of query is simple, it can potentially be extended to support a large range of statistical learning algorithms [9, 10].

3.2.3 Coin Pool Generation

In the step of differentially private noise addition (Step 4), we assume that there is a pool of unbiased coins (per each individual analyst) maintained at the proxy. In this section, we describe how to generate this pool of coins.

The most straightforward way would be to let the proxy take full responsibility for generating the coins. However, this would allow the curious proxy to know the true noise-free aggregate answer should the analyst choose to publish its noisy aggregate answer (see §4.3). As an alternative, the clients could generate the required unbiased coins and send them to the proxy. However, in the absence of an expensive protocol such as secure multi-player computation [26, 52], malicious clients could generate biased coins.

In our system, the proxy and clients generate the unbiased coins collaboratively (see Figure 2). Each online client itself *periodically* generates an encrypted unbiased coin $\mathscr{E}(o_c)$ with an analyst's public key, and sends it to the proxy.

Of course, malicious clients could still generate biased coins. To defend against this, the proxy does two things. First, when the proxy receives a client-generated

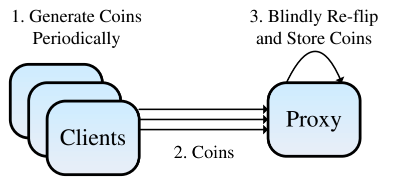

1. Generate Coins Periodically
3. Blindly Re-flip and Store Coins
Clients
Proxy
2. Coins

Figure 2: Unbiased Coin Generation

encrypted coin $\mathscr{E}(o_c)$, it verifies the legitimacy of the coin by checking its Jacobi symbol [18, 48] (as in Step 3). If this fails, the coin is dropped. Second, the proxy blindly *re-flips* the coin $\mathscr{E}(o_c)$ by multiplying it with the proxy's locally generated unbiased coin $\mathscr{E}(o_p)$, plus a modulo operation. This blind encrypted-coin re-flipping is possible because of the *XOR-homomorphic* feature of the GM cryptosystem we use: $\mathscr{E}(o_c) \times \mathscr{E}(o_p) \bmod m = \mathscr{E}(o_c \oplus o_p)$, where m is a part of the analyst's public key, and '$\oplus$' is the exclusive-or operator. This feature ensures that the modulus product $\mathscr{E}(o_c \oplus o_p)$ is an encrypted unbiased coin, regardless of any bias in the client's original coin. Note that the proxy does not know the actual (decrypted) value of the generated unbiased coin. Once generated, the proxy stores the unbiased coin in the locally maintained pool.

3.3 Practical Considerations

3.3.1 Utility of Aggregate Answer

The utility of the final aggregate answer learned by an analyst depends directly on the amount of added noise. The n coins added by the proxy (see expression 3), followed by the analyst's adjustment on the mean of $n/2$ (see expression 4), form a binomial distribution, which is a good approximation to the normal distribution $N(0, n/4)$.

The normal distribution is convenient in understanding the effect of added noise on the aggregate answer. In particular, the "68-95-99.7 rule" of the normal distribution tells us that the noisy aggregate answer falls within one, two, and three standard deviations away from the true noise-free answer with the probability of 68%, 95%, and 99.7%, respectively, where the standard deviation $\sigma = \sqrt{n}/2$. To give a concrete example, imagine that $c = 10^6$ clients, and a conservative privacy parameter of $\varepsilon = 1.0$ [39] is chosen. Given the normal distribution, for each bucket, there is a 68% probability that the noisy aggregate answer is within 15.24 away from the true answer, a 95% probability that it is within 30.48 away from the true answer, and a 99.7% probability that it is within 45.72. This gives high accuracy relative to the number of clients queried, and therefore the noisy aggregate answer has high utility even under a conservative privacy parameter setting.

3.3.2 Privacy Deficit vs. Privacy Budget

Under the definition of (ε, δ)-differential privacy, privacy loss is cumulative. This has given rise to the notion of a privacy budget [15, 17], where additional queries are simply not allowed after the cumulative ε and δ have reached some limit. This notion is completely impractical for our setting, where a user's personal data may persist for a lifetime or even longer. Whether or not to "kill" a database after some budget is reached is a policy decision. We prefer to treat the cumulative ε and δ as an ongoing measure of privacy loss rather than a hard limit (Step 5). As such, we refer to this measure as the *privacy deficit* rather than a privacy budget.

The assumption of cumulativity in differential privacy is very pessimistic because it effectively assumes that the correlation between answers to different queries is 100%, in other words, equivalent to repeating the same query (to the same statistical database or, in our case, to the same set of clients). While it could in theory be that every query is 100% correlated, in practice, many queries may not be very correlated. Furthermore, it is possible to informally estimate the amount of correlation between queries. For instance, the query "what percentage of users are male?" is highly correlated with the query "what percentage of users wear men's shoes?", but (probably) poorly correlated with the query "what percentage of users have spaghetti as their favorite food?".

Other factors also contribute to making queries not very correlated. In our setting, it is normally the case that only a fraction of clients actually answer a given query, and the set of clients changes from query to query. In addition, some of the data held at clients may change over time, making correlation between earlier and later queries less likely.

Given all the above, we can treat the privacy deficit as a *worst-case* measure of privacy loss. The actual privacy loss can be roughly estimated by taking into account the above factors.

Practical Approach. One way to leverage the privacy deficit would be to charge analysts accordingly, as suggested in [24]. In the case of PDDP, this charge would be a function of $c \times \varepsilon$, i.e., the total amount of increased privacy deficits across all queried clients. This would incentivize analysts to minimize the number of queries, the number of queried clients c, and the privacy parameter ε, while still obtaining high-utility answers. The best strategy for this depends on the application domain.

Note also that clients may locally maintain their own privacy deficits, and are free to not answer queries if some limit has been reached (or for any other reasons). Enabling this requires that the query sent from proxy to clients in Step 2 contain ε and δ (or c, if the value of δ is based on c).

3.3.3 Non-numeric Queries

Our SQL-style queries with associated buckets allow for a rich variety of sophisticated queries that produce numeric answers. Unfortunately, not all queries that an analyst may wish to make are numeric in nature. For instance, for the query "which website do you visit most often?", a client's answer may be one out of millions of websites. While non-numeric, this query can be transformed into a numeric query by mapping each website an analyst wishes to learn about into a numeric value. Unfortunately, this may produce a large number of buckets. This can often be partially mitigated in practice, for instance in this case by limiting the answer to the top 5000 most popular websites, as well as a "none of the above" bucket.

3.3.4 Sybils and NATs

The design, as described in §3.2, is susceptible to Sybil attacks [12], whereby a single client machine masquerades as multiple clients. In PDDP, the proxy can deal with this by limiting the number of clients selected at a single IP address for a given query. This may have the effect of biasing answers, for instance towards home users, where there are relatively few machines behind a NAT, and away from business users, where many users may be behind a NAT. This limit could be set by the analyst, thus giving it some control over this bias at the expense of a higher risk of answer skew by Sybil clients.

3.3.5 Scaling Considerations

To scale the PDDP proxy to millions of clients and queries, we propose a two-tier approach. The *front-end* devices at the proxy are responsible for individual clients. They exchange messages with clients and maintain the per-client states such as privacy deficits and last-connected timestamps. These devices scale easily through the addition of more front-end devices and the databases that support them. This is because there is no need for interaction between these devices other than the pairwise interaction needed for redundancy. An IP load-balancing router can be used to steer clients to the appropriate front-end devices.

The *back-end* devices at the proxy are responsible for individual queries. Each back-end device maintains a list of client IDs and the corresponding front-end devices that service these clients. This list does not need to be a complete list of all clients. It is enough if the list is a representative random selection of clients, but big enough to handle queries with a large target population. A query is submitted to a back-end device, which selects a set of clients, and passes the query to the appropriate front-end devices along with a list of clients to query. The back-end device will collect answers and coins, and add noise. As with front-end devices, the back-end devices do not require interaction beyond redundancy needs, and so also scale with the addition of more devices.

4 Privacy Analysis

In this section, we provide an analysis of the privacy properties of PDDP design given the threat assumptions from §2.

4.1 Analyst

The analyst can influence the content of the messages that it originates (i.e., "$query, \{L_i, U_i\}_{i=1}^{b}, c, \varepsilon$" in Step 1), and the messages that infected clients originate (i.e., "$\{v_i\}_{i=1}^{b}$" in Step 3).

4.1.1 Defending Against Malicious Analyst

An analyst could generate a query that attempts to reveal an individual client's information, for instance by specifying personally identifiable information (PII) as a predicate in the query. The noise added by the proxy defeats this (Step 4), at least within the guarantees given by differential privacy. Note that in most application domains, it may be perfectly reasonable to disallow PII as a predicate, thus giving stronger privacy protection in practice.

By enforcing a maximum limit of ε, the proxy prevents an analyst from giving a large ε in the query to minimize the added noise (Step 2). The analyst may select a small value of c in an attempt to isolate a single client. The differentially private noise defeats this (Step 4). Alternatively, the proxy can also simply enforce a lower bound on the value of c. In addition, the fact that the proxy selects random clients to query prevents an analyst from knowing which clients answered (policy 1 in Step 2). The session encryption between proxy and clients prevents an eavesdropping analyst from knowing which client received which query (§3.2.1). If necessary, the proxy can add delay or chaff to the queries sent to clients, in order to prevent an eavesdropping analyst from identifying the queried clients by using traffic analysis to correlate the query sent to the proxy with the subsequent messages sent to clients.

On the assumption that clients have the proxy's correct public key, the session encryption prevents the analyst from becoming a man in the middle between clients and proxy (§3.2.1). Similarly, the analyst as an eavesdropper cannot see the answers generated by the clients, and so cannot trivially decrypt them and associate answers with clients. The analyst also cannot, through traffic analysis, correlate answers received by the proxy to those sent by the proxy. This is because the proxy stores, delays, and shuffles all answers and coins before sending them to the analyst (Step 5).

4.1.2 Defending Against Clients Infected by Analyst

The analyst may try to create a covert channel between itself and infected clients by manipulating client answers. However, the analyst cannot create a covert channel by having the infected client directly embed it into the transmitted ciphertext. This is because the resulting ciphertext would not satisfy the Jacobi symbol checking made by the proxy (Step 3), and so would be discarded. The analyst also cannot create a covert channel by creating a word from per-bucket values in the full answer $\{v_i\}_{i=1}^{b}$. This is because all the answer and noise values within each individual bucket are mixed and shuffled at the proxy (Step 5). Note that this also prevents the identification of an individual client through correlation across per-bucket answer values.

In the GM bit-cryptosystem, the encryption algorithm $\mathscr{E}$ takes one bit $w \in \{0,1\}$ and the public key (x,m) as input, and outputs the ciphertext $\mathscr{E}(w) = r^2 x^w \pmod{m}$, where r is a random number from $\mathbb{Z}_m^*$. Without the fix introduced at the end of this paragraph, the analyst could create a covert channel by knowing the sequence of random numbers r that a given client uses to produce the ciphertexts. This could be done for instance by having the client use the same known random number for each answer bit, or by knowing the seed for the random number generator. This would allow the analyst to predict which ciphertext would be produced for the subsequent '1' or '0' from the client, thus allowing it to isolate that client's answer stream from those of other clients. This covert channel can easily be destroyed by having the proxy *re-randomize* every client-produced answer value by homomorphically XOR-ing it with a randomly encrypted '0'. This effectively scrambles the ciphertext without modifying the client's answer.

4.2 Client

An adversary may try to distort the final aggregate answer by creating clients that produce incorrect answers. Because of the use of GM bit-cryptosystem, individual clients are strictly limited to generate only a single bit per bucket. Therefore, the *maximum* absolute distortion per bucket in the final aggregate answer is bounded by the number of malicious clients (here ignoring distortion due to the differentially private noise). This is in contrast with previous distributed differential privacy designs [30, 45, 49], where even a *single* malicious client can substantially distort the final aggregate answer.

Our system thus raises the bar for the adversary by forcing it to deploy a large number of malicious clients, especially when limits are placed on the number of clients that can answer a query from a single IP address, as described in §3.3.4. Deploying a large number of malicious clients increases the likelihood that the nature of the client attack will be detected, thus allowing the analyst to at least know which queries are incorrect. What's more, even when there are a large number of malicious clients, due to the noise added by the proxy, our system *always* provides differentially private guarantees at least for every honest client.

4.3 Proxy

There are many legitimate cases where an analyst may wish to publish the (noisy) aggregate answers that it learns. Because the honest-but-curious proxy does not know the actual value of the coins that it uses to add noise (see §3.2.3), it cannot determine the true noise-free aggregate answer by subtracting the added noise from the aggregate answer published by the analyst.

Note that it would be trivial for a *dishonest* proxy to generate the required unbiased coins by itself (i.e., not base them on a re-flipping of the client-generated coins). This could not be detected because the correctly generated coins in any event appear as random perturbations on the original client-generated coins. Were this dishonest proxy a concern in practice, we could require that the analyst itself add differentially private noise to final aggregate answers before publishing them. Of course, a malicious analyst may choose not to add this noise. Absent collusion, however, the proxy would not know which analysts were adding noise, and which were not.

5 Implementation and Evaluation

5.1 Implementation and Deployment

We implemented a fully functional PDDP system. Following our design in §3.2, the implementation comprises three basic components: client, proxy, and analyst.

The **client** is implemented as a Firefox add-on (as shown in Figure 3). It consists of 9600 lines of Java code, compiled into JavaScript using the Google Web Toolkit. It captures users' web browsing activities (such as webpages visited, searches made, etc.), certain online shopping activities (such as how often items are placed in shopping carts), and certain ad interactions (such as the number of ads viewed and clicked). In principle, our Firefox add-on can be extended to capture any online activities made by the user. All captured information is stored in a local SQLite database using Firefox's Storage API, thus allowing our system to query for that information. Every 5 minutes, the add-on connects to the proxy to retrieve any pending queries, and return answers and periodically generated coins.

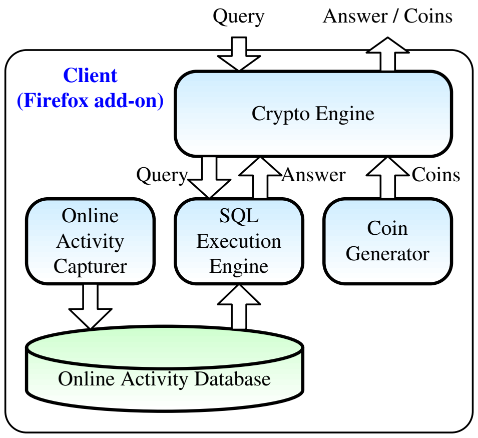


Figure 3: Client implementation

The **proxy** is implemented with 3600 lines of code as a web server running Tomcat 6.0.33. It forwards the queries from analyst to clients, receives client-generated answers and coins, adds differentially private noise, and sends answers with noise back to the analyst. Proxy state is stored in a back-end MySQL database.

The **analyst** is implemented in 800 lines of Java code.

After verifying the correctness of PDDP on a set of local machines (where we have access to the local data), we deployed PDDP on more than 600 actual clients. This deployment allows us to make queries to exercise and test our system.

5.2 Comparison: An Alternative Design

In our evaluation, we compare PDDP with a strawman design that is more straightforward and might at first glance appear to be a natural choice. Indeed, this was one of the designs that we initially focused on. Like [45], it exploits the additive homomorphism of the Paillier cryptosystem [42]. Like PDDP, however, it uses an honest-but-curious proxy in order to avoid expensive protocols like the distributed key distribution in [45].

In the "Paillier-based" design, each queried client returns an encrypted 1 or 0 in each bucket. Due to Paillier's additive homomorphism, the proxy can directly sum up all clients' encrypted answers to get the encrypted total sum for each bucket. The proxy adds the required differentially private noise to each bucket to generate the encrypted noisy aggregate answer, and then forwards this answer to the analyst. The analyst uses its private key to decrypt the noisy aggregate answer.

Like [45], this basic design suffers from the problem that a single malicious client can provide an answer value other than 1 or 0 in each bucket to substantially distort the

Table 1: Performance at client (sustained crypto operations per second)

	Encryption			ZKP Generation		
	Firefox	Chrome	Smartphone	Firefox	Chrome	Smartphone
PDDP System	2157.96	22773.86	808.87	–	–	–
Paillier-based System	0.59	7.79	0.27	0.17	2.34	0.08

Table 2: Performance at proxy and analyst (sustained crypto operations per second). Homomorphic operations are XOR in the PDDP system, and addition in the Paillier-based system.

	Encryption	Decryption	Homomorphic Op	ZKP Generation	ZKP Verification
PDDP System	15323.32	6601.10	123609.39	–	–
Paillier-based System	62.76	63.41	34188.03	18.70	15.58

aggregate answer without detection. To overcome this problem, we used non-interactive zero-knowledge proofs (ZKP) [8] based on the Fiat-Shamir heuristic [23] to ensure that all encrypted answer values are either 1 or 0. The resulting design is almost apples-to-apples comparable with PDDP. The primary difference, however, is that in the Paillier-based design, the proxy knows exactly how much noise has been added, and therefore knows the true aggregate answer in those cases where the analyst publishes its noisy aggregate answer (see §4.3).

5.3 Evaluation

5.3.1 Computational Overhead

A major concern in the performance of distributed differentially private systems is the overhead of the asymmetric crypto operations. In this section, we measure the crypto performance of PDDP, as well as compare it with the Paillier-based system outlined in §5.2.

To generate an answer to a received query, each client needs to encrypt a binary value for each bucket, using the GM cryptosystem in PDDP or using the Paillier cryptosystem in a Paillier-based system. We measure the sustained rate at which three different clients can do these crypto operations: a Firefox browser and a Chrome browser on a desktop workstation running Linux 2.6.32 with Intel dual core 3GHz, and a WebKit browser on a smartphone running Android 2.2 with a 1GHz processor. The results, with a 1024-bit key length, are given in Table 1. The PDDP crypto operations are very fast. Even the smartphone can execute over 800 encryptions per second. This suggests that crypto is not a bottleneck in PDDP's client implementation, though in practice these operations should be run in the background for queries with many buckets. By contrast, the Paillier-based clients are prohibitively slow: 8 encryptions per second with Chrome, and less than one per second for the smartphone. If zero-knowledge proofs (ZKPs) are used to ensure that clients produce answers with only 1's and 0's (see §5.2), Paillier-based clients are even slower.

At the proxy, the PDDP system needs to perform GM crypto-operations to transform periodically client-generated coins into unbiased coins — one (offline) encryption and one homomorphic XOR for one unbiased coin. Moreover, the PDDP proxy needs to perform the Jacobi symbol checking on received coins and answer values. This checking cost is the same as (or faster than) the cost of GM decryption. Table 2, also with a 1024-bit key length, shows that all these crypto operations are quite efficient in PDDP. In a Paillier-based system, the proxy needs to first verify all clients' submitted ZKPs — one ZKP for each client answer's each bucket. The ZKP verification is very expensive as shown in Table 2. Furthermore, the proxy in a Paillier-based system needs to homomorphically sum up all client answers for each bucket, and then add locally generated differentially private noise to each per-bucket total sum.

To give a concrete example, we assume a 10-bucket query sent to 1M clients, with a conservative privacy parameter of $\varepsilon = 1.0$ [39]. Based on the results in Table 2, the PDDP proxy would require less than 1 CPU-second for the GM crypto-operations, and less than 30 CPU-minutes for the Jacobi symbol checking. The Paillier-based proxy, by contrast, would require roughly 5 CPU-minutes for the homomorphic additions, and roughly 1 CPU-week for the zero-knowledge proof verifications.

At the analyst, while the PDDP system needs to decrypt all encrypted values (i.e., all client answers plus coins) within each bucket, the Paillier-based system only needs to decrypt one encrypted value within each bucket. The computational overhead at the PDDP analyst is higher than at the Paillier-based analyst. Nevertheless, the overhead is very reasonable considering the efficiency of PDDP's crypto operations.

5.3.2 Bandwidth and Storage Overhead

In both PDDP and Paillier-based systems, a client needs to transmit an encrypted answer value for each bucket. A

PDDP client also needs to transmit a periodically generated coin to the proxy, while a Paillier-based client needs to transmit a ZKP for each bucket of an answer. The bandwidth requirements for these are modest: on the order of a few kilobytes.

The PDDP proxy needs to store all queried clients' encrypted answer values for each bucket. Assuming a key length of 1024 bits, the storage for a 10-bucket query over 1 million clients is about 1.2GB. These answers can be stored on a hard-drive. The PDDP proxy also needs to receive, re-flip, and store client-generated coins. The required number of coins, however, is only a small fraction of the number of answer values. As an example, for the 10-bucket, 1M-client, $\varepsilon = 1.0$ query, only 9290 coins are required (929 per bucket). Finally, the answers and coins will be transmitted to the analyst (roughly 1.2GB), which stores and decrypts them. Overall, the storage and bandwidth requirements for PDDP proxy are quite reasonable. Recall that it is straightforward to further scale the system by adding proxy devices (see §3.3.5).

Note that the Paillier-based proxy and analyst have minimal storage and bandwidth requirements because they need to store and transmit only one aggregate answer value per bucket. Each time the Paillier-based proxy receives an answer from a client, it adds the per-bucket answer value to the corresponding bucket of the locally maintained aggregate answer.

In summary, the PDDP system overhead is within very reasonable limits for all system components. Compared with a Paillier-based system, the PDDP system trades-off moderate and affordable bandwidth and storage overhead against unacceptable computational overhead.

5.3.3 Querying the Client Deployment

Before we deployed PDDP at scale, we first tested it on a set of local machines. Since we had access to the data on the local machines, we knew the true answers of the clients running on those local machines. In so doing, we were able to verify the correctness of PDDP. We then deployed PDDP on more than 600 clients taken from "friends and family" as well as Amazon Mechanical Turk. This deployment allowed us to make queries to exercise and test the PDDP system.

Figure 4 gives a histogram of the number of connections received from clients that established at least one connection on Sep. 26. While the Firefox browser is active, clients make connections every 5 minutes. We see that only one or two percent of client browsers were running the full 24 hours. Almost half of the clients made 20 or fewer connections, representing a browser uptime of less than 2 hours in 24. This suggests that it could easily take several days to complete a query over nearly all of a set of clients. It also suggests that an approach

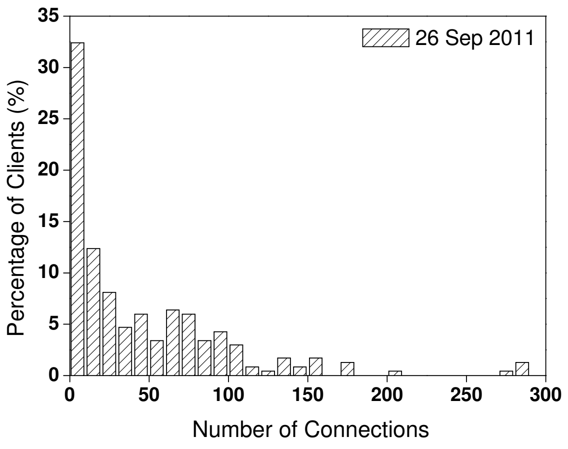


Figure 4: Number of connections received from clients on Sep. 26, 2011

that requires that clients are continuously available, as with [45], will not work in our setting.

To further test our system, we executed a number of PDDP queries over our deployed client base. Each query sets $c = 250$ (out of over 600 clients) and $\varepsilon = 5.0$. Each query lasts 24 hours, and we select clients as they connect until $c = 250$ unique clients are queried or 24 hours expires, whichever comes first. Finally, each query covers statistics gathered over a 24 hour period — midnight-to-midnight of the day specified in the query in the client's local time.

Note that what we report here are the noisy aggregate answers from our deployment. To preserve users' privacy, PDDP didn't and indeed shouldn't output the true noise-free aggregate answers. This is our key premise and motivation — learn statistical results without knowing users' true answers. Nevertheless, our experimental parameter settings (i.e., $c = 250$ and $\varepsilon = 5.0$) result in 16 coins per bucket, and ensure that a per-bucket aggregate answer is within plus or minus 2, 4, and 6 of the true noise-free aggregate answer, with the probability of 68%, 95%, and 99.7%, respectively (see §3.3.1).

We started by issuing the simple query "how many webpages did a client visit per day?" over five consecutive days. We specified 8 buckets as shown in Table 3. Between 176 and 230 answers were gathered each day (including noise). Since each client supplies a count in a single bucket only, this directly reflects (noisily) the number of unique clients that replied in 24 hours. The results in Table 3 show that the requested information can usefully be gathered through the PDDP system. For instance, we see that a large number of clients show very little activity (0∼10 webpages visited). What's more, the number of these low-activity clients was greater during the weekend (Sep. 24 and 25). On the other hand, a

Table 3: Number of webpages visited per day (displayed as noisy client counts within each bucket)

	0∼10 pages	11∼20 pages	21∼50 pages	51∼100 pages	101∼200 pages	201∼500 pages	501∼1000 pages	>1000 pages	Total
24 Sep 2011	73	1	10	14	29	33	10	6	176
25 Sep 2011	115	3	14	11	30	43	17	-3	230
26 Sep 2011	61	20	16	21	44	48	12	3	225
27 Sep 2011	49	8	15	17	25	44	20	2	180
28 Sep 2011	56	7	8	19	27	45	16	3	181

Table 4: Five different queries towards clients' activities on Sep. 29, 2011 (displayed as noisy client counts within each bucket)

	0	1∼10	11∼20	21∼50	51∼100	101∼200	201∼500	501∼1000	>1000	Total
Q_1	45	11	5	18	23	24	37	18	4	185
Q_2	46	40	28	45	19	7	0	0	-3	182
Q_3	43	17	9	23	24	30	33	2	2	183
Q_4	56	32	20	28	31	8	6	-1	2	182
Q_5	77	28	17	25	8	16	6	5	1	183

Q_1: How many webpages visited?
Q_2: How many unique websites visited?
Q_3: How many visited webpages on a user's top 3 favorite websites?
Q_4: How many searches made?
Q_5: How many Google ads shown?

substantial number of clients visit between 100 and 500 webpages per day. Note that in one case the added noise produces a negative count. This poses no privacy risk; if desired, an analyst can safely replace all negative counts with zeros without privacy loss [33].

We additionally issued five queries towards clients' activities on Sep. 29, as listed in Table 4. We also added a '0' bucket in order to distinguish between truly idle clients and low-activity clients. Once again, this set of queries shows that it is possible to gather meaningful noisy data from even a relatively small population of clients. For instance, we see from query Q_1 that indeed a large fraction of low-activity browsers are idle (45 clients in '0' bucket). We see from query Q_2 that a relatively large number of users (40 of 136 non-idle clients) visited a small number of unique websites (between 1 and 10). Moreover, given the rough similarity between the results of queries Q_1 and Q_3, it appears that most browsing takes place over users' top 3 favorite websites. Many additional observations can be made; for instance, from queries Q_4 and Q_5 we see that search is often used (around half of the clients made 11∼100 searches), and that Google ads are shown relatively often.

6 Malicious Proxy

In this section, we sketch out how we may weaken proxy trust requirements through the use of trusted hardware such as the Trusted Platform Module (TPM) or the IBM 4758 cryptographic coprocessor [44]. In particular, we assume that the proxy is malicious in that it may try to violate the privacy of individual clients, and is willing to collude with analysts (or pose as an analyst). On the other hand, we assume that the proxy is unwilling or unable to physically tamper with the hardware, including placing probes on the system bus in order to record system operations, for instance because there is a threat of random inspections from a third party.

The basic idea is that the proxy runs an executable which has been verified by a trusted third party to operate correctly as a proxy. This executable is then remotely attested by the clients, as described in [44].

There are two attacks that we need to defend against. First, the proxy may try to de-anonymize a client by associating a given answer with the client's IP address, and conveying this association to the analyst. Second, the proxy may try to avoid adding noise to the answers, thus violating differential privacy. We assume that the proxy operator is able to launch a reboot attack on the proxy hardware. In this attack, after a client attests the executable and transmits its answer to the proxy, the proxy is rebooted with a malicious executable which carries out one of the above attacks. Prior work has shown how to prevent a reboot attack once the client maintains a secure channel with the proxy after attestation [25, 37]. In our scenario, however, a client may drop the secure channel after it has supplied its answer (Step 3) but before the proxy transmits the answer to the analyst (Step 5).

In our design, we assume that after a client connects to the proxy, the query is received and answered, as well as any coins are delivered, within the same attested secure channel. The proxy operates as follows:

- When the proxy receives a coin from a client, it re-flips the coin and stores it into the unbiased coin pool in memory. Upon booting, previous stored coins will be lost and the pool will be empty. Therefore, the proxy will have to gather a modest number of coins (some thousands) before it can start handling queries from analysts.

- When the proxy receives a query from the analyst, it immediately reads the required number of coins from the unbiased coin pool, and places them in a newly initialized linked list or similar data structure. If there are not enough unbiased coins remaining in the pool, the proxy waits until enough coins have been generated.

- When the proxy receives a (periodic) connect request from a client with a pending query, it sends the query to the client and receives an answer. The answer is placed in the linked list at a random location, and no association with the client answer is established.

- When enough client answers have been received, all items in the linked list (answers and coins mixed together) are transmitted to the analyst.

Analysis. By requiring that the proxy read the coins before receiving any client answers, we ensure that even a single client answer is adequately mixed with noise. This prevents a client de-anonymization attack whereby the proxy is rebooted with a malicious executable after only a single answer is received, thus allowing the malicious executable to link the received answer to the client by recording network activity. This also prevents the no-noise attack, by ensuring that noise is mixed in from the very beginning, and cannot be distinguished from client answers. Further work is required to fully analyze this approach to accommodating a malicious proxy.

7 Related Work

To preserve user privacy, early approaches like anonymization sanitize the user data by removing well-known personally identifiable information (PII), e.g., name, gender, birthday, social security number, ZIP code, etc. These approaches not only heavily restrict the utility of the user data [43], but ultimately cannot effectively protect user privacy [11, 41, 46]. In a well-publicized example of this, an individual user was identified from AOL's anonymized search logs [1]. A number of privacy-preserving approaches have been proposed [5, 6, 21, 22, 35, 36, 50], discussed in the following.

One general approach [5, 6] is to randomize user data by adding random distortion values drawn independently from some known distribution such as a uniform distribution or a normal distribution. An analyst can collect the distorted user data and reconstruct the distribution of the original data using for instance the expectation maximization (EM) algorithm [5]. However, [22] indicates that such simple randomization is potentially vulnerable to privacy breaches. Evfimievski *et al.* then proposed a class of new randomization operators [22], and used them to limit the privacy breaches [21].

k-anonymity [50] ensures that each individual is indistinguishable from at least $k-1$ other individuals with respect to certain sensitive attributes, thus individuals cannot be uniquely identified. While k-anonymity protects against identity disclosure, it does not prevent attribute disclosure if there is little diversity in these sensitive attributes. To address this problem, l-diversity [36] was proposed which requires that there exist at least l well-represented values for each sensitive attribute. Recently, [35] indicated that l-diversity may not be necessary, and it is also insufficient to prevent attribute disclosure. To solve the problems of both k-anonymity and l-diversity, [35] further proposed t-closeness which requires that the distribution of a sensitive attribute in any "equivalence class" is close to the overall distribution of the attribute, and the distance between the two distributions is no more than a threshold t.

Many of these approaches, however, only provide syntactic guarantees on the privacy, and impose constraints of one sort or another. Differential privacy [13, 14, 17] is considered stronger than previous approaches because it provides a provable guarantee that it is hard to detect the presence or absence of any individual records, as already discussed in §3.1. Differential privacy does not make any assumptions about the adversary. It is independent of the adversary's computational power and auxiliary information. However, original differential privacy mechanisms were not designed to support a distributed environment.

In theory, secure multi-player computation [26, 52] could be used to emulate differential privacy in a distributed manner. It would be, however, highly expensive. To our knowledge, there are four previous designs for distributed differential privacy [16, 30, 45, 49]. None of them are realistically practical in a large-scale distributed setting.

The first distributed differential privacy design was proposed by Dwork *et al.* in [16]. It is something of a cryptographic tour de force, variously exploiting Byzantine agreement, distributed coin flipping, verifiable secret sharing, secure function evaluation, and randomness extractor. However, its computational load per user is $O(U)$, where U is the number of users, making it impractical for a large-scale setting.

To reduce this complexity, Rastogi and Nath in [45] designed a two-round protocol based on the threshold Paillier cryptosystem, and Shi *et al.* in [49] designed a single-round protocol based on the decisional Diffie-Hellman assumption. Both designs [45, 49] achieve distributed differential privacy while reducing the computational load per user from $O(U)$ to $O(1)$. However, their distributed key distribution protocols cannot work at large-scale under churn. To solve this problem, Götz and Nath in [30] utilized two honest-but-curious servers to collaboratively collect users' noisy answers and compute the final aggregate result. Nevertheless, in all of these designs [30, 45, 49], even a single malicious user can substantially distort the final result without detection.

8 Conclusion and Future Work

In presenting a practical system, PDDP, that supports statistical queries over distributed private user data, this paper takes a first step towards practically mining that data while preserving privacy. Still, there is a gap between the utility of a central database and that of PDDP. While one might be willing to give up some utility in exchange for privacy, it is important to maximize the utility of distributed private user data.

Previous work shows that it is possible to support various statistical learning algorithms through a differentially private interface [9, 10]. These algorithms, however, require a sequence of queries to a given database (or, in our case, a given set of clients). It is not clear whether this is practical over PDDP. While in principal the proxy may retain state about which set of clients should receive a sequence of queries, in practice some fraction of these clients will become unavailable temporarily or permanently during the course of the queries. One avenue of future work is to understand the impact of this on the utility of the data as well as the time it takes to run these algorithms.

A second area of future work concerns the scalability of non-numeric queries (§3.3.3). A brute-force approach of assigning a numeric value to every possible non-numeric answer does not scale for certain queries, e.g., "what is the most frequent search?". One possible approach might be to use invertible Bloom filters [20, 29] to map a large number of possible non-numeric answers into a relatively small number of buckets, at some loss of fidelity. However, we need to understand the trade-off between utility and scalability. An interesting question is whether this loss of fidelity itself can be exploited as a privacy mechanism.

More broadly, we need to gain experience with PDDP. Towards this end, we plan to use PDDP to gather statistical data for a large-scale experiment with the Privad privacy-preserving advertising system [31]. By "eating our own dog food", we can learn first-hand the limitations of PDDP in gathering meaningful data from clients.

While we believe that it is reasonable in practice to require an honest-but-curious proxy, it would obviously be better not to. One avenue of future work is to nail down the design and security properties of a TPM-based approach as sketched in §6. Ultimately, a challenging open problem is to design a system that practically provides differentially private statistical queries over distributed data without requiring a trusted third party.

Finally, the question of measuring actual privacy loss (versus worst-case privacy loss) is extremely important for differential privacy, not only in a distributed setting but generally. While we assume that this necessarily involves incorporating domain-specific information such as user sensitivity and adversarial knowledge, and therefore gives up some of the rigor of differential privacy, we believe it is important to make progress here in order to make differential privacy more practical.

9 Acknowledgments

We are grateful to the anonymous reviewers and our shepherd, Nickolai Zeldovich, for their insightful comments. We also thank Rose Hoberman, Aniket Kate, Stevens Le Blond, and Nan Zheng for their valuable feedback on drafts of this work. Finally, we would like to thank the volunteers for anonymously helping us exercise our system.

References

[1] A Face Is Exposed for AOL Searcher No. 4417749. `http://www.nytimes.com/2006/08/09/technology/09aol.html`.

[2] FTC Charges Deceptive Privacy Practices in Google's Rollout of Its Buzz Social Network. `http://www.ftc.gov/opa/2011/03/google.shtm`.

[3] Microsoft HealthVault. `http://www.microsoft.com/en-us/healthvault/`.

[4] Project VRM. `http://cyber.law.harvard.edu/projectvrm/Main_Page`.

[5] AGRAWAL, D., AND AGGARWAL, C. C. On the Design and Quantification of Privacy Preserving Data Mining Algorithms. In *PODS* (2001).

[6] AGRAWAL, R., AND SRIKANT, R. Privacy-Preserving Data Mining. In *SIGMOD Conference* (2000), pp. 439–450.

[7] BACKES, M., KATE, A., MAFFEI, M., AND PECINA, K. ObliviAd: Provably Secure and Practical Online Behavioral Advertising. In *IEEE Symposium on Security and Privacy* (2012).

[8] BAUDRON, O., FOUQUE, P.-A., POINTCHEVAL, D., STERN, J., AND POUPARD, G. Practical multi-candidate election system. In *PODC* (2001), pp. 274–283.

[9] Blum, A., Dwork, C., McSherry, F., and Nissim, K. Practical privacy: the SuLQ framework. In *PODS* (2005), pp. 128–138.

[10] Blum, A., Ligett, K., and Roth, A. A learning theory approach to non-interactive database privacy. In *STOC* (2008), pp. 609–618.

[11] Coull, S. E., Wright, C. V., Monrose, F., Collins, M. P., and Reiter, M. K. Playing Devil's Advocate: Inferring Sensitive Information from Anonymized Network Traces. In *NDSS* (2007).

[12] Douceur, J. R. The Sybil Attack. In *IPTPS* (2002), pp. 251–260.

[13] Dwork, C. Differential Privacy. In *ICALP* (2006), pp. 1–12.

[14] Dwork, C. Differential Privacy: A Survey of Results. In *TAMC* (2008), pp. 1–19.

[15] Dwork, C. A firm foundation for private data analysis. *Commun. ACM 54*, 1 (2011), 86–95.

[16] Dwork, C., Kenthapadi, K., McSherry, F., Mironov, I., and Naor, M. Our Data, Ourselves: Privacy Via Distributed Noise Generation. In *EUROCRYPT* (2006), pp. 486–503.

[17] Dwork, C., McSherry, F., Nissim, K., and Smith, A. Calibrating Noise to Sensitivity in Private Data Analysis. In *TCC* (2006), pp. 265–284.

[18] Eikenberry, S. M., and Sorenson, J. Efficient Algorithms for Computing the Jacobi Symbol. In *ANTS* (1996), pp. 225–239.

[19] Enck, W., Gilbert, P., gon Chun, B., Cox, L. P., Jung, J., McDaniel, P., and Sheth, A. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *OSDI* (2010), pp. 393–407.

[20] Eppstein, D., and Goodrich, M. T. Straggler Identification in Round-Trip Data Streams via Newton's Identities and Invertible Bloom Filters. *IEEE Trans. Knowl. Data Eng. 23*, 2 (2011), 297–306.

[21] Evfimievski, A. V., Gehrke, J., and Srikant, R. Limiting privacy breaches in privacy preserving data mining. In *PODS* (2003), pp. 211–222.

[22] Evfimievski, A. V., Srikant, R., Agrawal, R., and Gehrke, J. Privacy preserving mining of association rules. In *KDD* (2002), pp. 217–228.

[23] Fiat, A., and Shamir, A. How to Prove Yourself: Practical Solutions to Identification and Signature Problems. In *CRYPTO* (1986), pp. 186–194.

[24] Ghosh, A., and Roth, A. Selling privacy at auction. In *ACM Conference on Electronic Commerce* (2011), pp. 199–208.

[25] Goldman, K., Perez, R., and Sailer, R. Linking remote attestation to secure tunnel endpoints. In *STC* (2006), pp. 21–24.

[26] Goldreich, O., Micali, S., and Wigderson, A. How to Play any Mental Game or A Completeness Theorem for Protocols with Honest Majority. In *STOC* (1987), pp. 218–229.

[27] Goldwasser, S., and Micali, S. Probabilistic Encryption and How to Play Mental Poker Keeping Secret All Partial Information. In *STOC* (1982), pp. 365–377.

[28] Goldwasser, S., and Micali, S. Probabilistic Encryption. *J. Comput. Syst. Sci. 28*, 2 (1984), 270–299.

[29] Goodrich, M. T., and Mitzenmacher, M. Invertible Bloom Lookup Tables. *CoRR abs/1101.2245* (2011).

[30] Götz, M., and Nath, S. Privacy-Aware Personalization for Mobile Advertising. In *Microsoft Research Technical Report MSR-TR-2011-92* (2011).

[31] Guha, S., Cheng, B., and Francis, P. Privad: Practical Privacy in Online Advertising. In *NSDI* (2011).

[32] Haeberlen, A., Pierce, B. C., and Narayan, A. Differential privacy under fire. In *USENIX Security Symposium* (2011).

[33] Korolova, A., Kenthapadi, K., Mishra, N., and Ntoulas, A. Releasing search queries and clicks privately. In *WWW* (2009), pp. 171–180.

[34] Krishnamurthy, B., and Wills, C. E. On the leakage of personally identifiable information via online social networks. In *WOSN* (2009), pp. 7–12.

[35] Li, N., Li, T., and Venkatasubramanian, S. t-Closeness: Privacy Beyond k-Anonymity and l-Diversity. In *ICDE* (2007), pp. 106–115.

[36] Machanavajjhala, A., Gehrke, J., Kifer, D., and Venkitasubramaniam, M. l-Diversity: Privacy Beyond k-Anonymity. In *ICDE* (2006).

[37] McCune, J. M., Parno, B., Perrig, A., Reiter, M. K., and Isozaki, H. Flicker: an execution infrastructure for tcb minimization. In *EuroSys* (2008), pp. 315–328.

[38] McSherry, F. Privacy integrated queries: an extensible platform for privacy-preserving data analysis. In *SIGMOD Conference* (2009), pp. 19–30.

[39] McSherry, F., and Mahajan, R. Differentially-private network trace analysis. In *SIGCOMM* (2010), pp. 123–134.

[40] McSherry, F., and Mironov, I. Differentially Private Recommender Systems: Building Privacy into the Netflix Prize Contenders. In *KDD* (2009), pp. 627–636.

[41] Narayanan, A., and Shmatikov, V. Robust De-anonymization of Large Sparse Datasets. In *IEEE Symposium on Security and Privacy* (2008), pp. 111–125.

[42] Paillier, P. Public-Key Cryptosystems Based on Composite Degree Residuosity Classes. In *EUROCRYPT* (1999), pp. 223–238.

[43] Pang, R., Allman, M., Paxson, V., and Lee, J. The devil and packet trace anonymization. *Computer Communication Review 36*, 1 (2006), 29–38.

[44] Parno, B., McCune, J. M., and Perrig, A. Bootstrapping Trust in Commodity Computers. In *IEEE Symposium on Security and Privacy* (2010), pp. 414–429.

[45] Rastogi, V., and Nath, S. Differentially private aggregation of distributed time-series with transformation and encryption. In *SIGMOD Conference* (2010), pp. 735–746.

[46] Ribeiro, B. F., Chen, W., Miklau, G., and Towsley, D. F. Analyzing Privacy in Enterprise Packet Trace Anonymization. In *NDSS* (2008).

[47] Roy, I., Setty, S. T. V., Kilzer, A., Shmatikov, V., and Witchel, E. Airavat: Security and Privacy for MapReduce. In *NSDI* (2010), pp. 297–312.

[48] Shallit, J., and Sorenson, J. A binary algorithm for the Jacobi symbol. *ACM SIGSAM Bulletin 27*, 1 (1993), 4–11.

[49] Shi, E., Chan, T.-H. H., Rieffel, E. G., Chow, R., and Song, D. Privacy-Preserving Aggregation of Time-Series Data. In *NDSS* (2011).

[50] Sweeney, L. k-Anonymity: A Model for Protecting Privacy. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems 10*, 5 (2002), 557–570.

[51] Toubiana, V., Narayanan, A., Boneh, D., Nissenbaum, H., and Barocas, S. Adnostic: Privacy Preserving Targeted Advertising. In *NDSS* (2010).

[52] Yao, A. C.-C. Protocols for Secure Computations. In *FOCS* (1982), pp. 160–164.

Koi: A Location-Privacy Platform for Smartphone Apps

Saikat Guha
Microsoft Research India
saikat@microsoft.com

Mudit Jain
Microsoft Research India
t-muditj@microsoft.com

Venkata N. Padmanabhan
Microsoft Research India
padmanab@microsoft.com

Abstract — With mobile phones becoming first-class citizens in the online world, the rich location data they bring to the table is set to revolutionize all aspects of online life including content delivery, recommendation systems, and advertising. However, user-tracking is a concern with such location-based services, not only because location data can be linked uniquely to individuals, but because the low-level nature of current location APIs and the resulting dependence on the cloud to synthesize useful representations virtually guarantees such tracking.

In this paper, we propose *privacy-preserving location-based matching* as a fundamental platform primitive and as an alternative to exposing low-level, latitude-longitude (lat-long) coordinates to applications. Applications set rich location-based triggers and have these be fired based on location updates either from the local device or from a remote device (e.g., a friend's phone). Our Koi platform, comprising a privacy-preserving matching service in the cloud and a phone-based agent, realizes this primitive across multiple phone and browser platforms. By masking low-level lat-long information from applications, Koi not only avoids leaking privacy-sensitive information, it also eases the task of programmers by providing a higher-level abstraction that is easier for applications to build upon. Koi's privacy-preserving protocol prevents the cloud service from tracking users. We verify the non-tracking properties of Koi using a theorem prover, illustrate how privacy guarantees can easily be added to a wide range of location-based applications, and show that our public deployment is performant, being able to perform 12K matches per second on a single core.

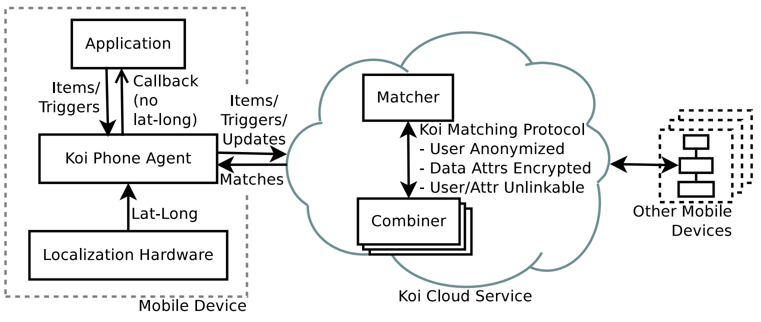
1 Introduction

The skyrocketing popularity of smart-phones has all but eviscerated the notion of "location-based services" as a separate class of applications. Today, virtually *all* applications and services must leverage location information. These applications and services include search, social networking, and multi-player games, to name just a few. Juxtaposed with this broad need for location information is the inherent complexity of individual applications operating on location data and its implications on user privacy. Recent high-profile incidents have put the spotlight on location privacy, with even governments and regulatory bodies asking questions of technology providers [6].

A popular approach in the research literature to providing location privacy is *obfuscation*, wherein a user's location coordinates are made imprecise by adding noise or are entirely suppressed, based on such factors as user density and the sensitivity of a location [19, 23]. While this approach has merits, obfuscation creates an unnecessary tension between the quality of location-based functionality and user privacy. Moreover, this approach does not address the crux of the privacy challenge: trusted applications (e.g., navigation app), which have a legitimate need for access to user location information, inadvertently leaking this to third-parties (e.g., Google maps), who are then in a position to link a user's ID to their location, and possibly track the user over time.

In this paper, we argue for a different approach, Koi, which is based on one key idea: switching to location *matching* instead of location *lookups*. In other words, rather than having an application look up mobile node's lat-long coordinates, Koi lets the application specify a location event of interest (e.g., proximity to fixed location such as a grocery store or the dynamic location of a friend) and notifies the application when there is a location match. Not only does the Koi approach relieve the application developer from having to work with the nitty-gritty of low-level lat-long information, it avoids polluting the application with lat-long information, thereby preventing accidental leakage of this information. Indeed, privacy incidents in the past have arisen from the carelessness (e.g., a third-party library used by the app developer, such as an advertising control, that constantly sends the user's lat-long to the third-party when lat-long is irrelevant to the original application [14]) rather than malice (e.g., an application that actively tries to subvert the location privacy of the mobile user).

The design of Koi comprises three main elements. The first, which arises directly from the approach outlined above and builds on prior concepts of triggers and symbolic locations [4, 30], is a *callback-based matching API* for applications. This API allows the application to specify the location event of interest, whether in terms of a static location or a dynamic location. The second element is a *privacy-preserving cloud-based matching service*, which uses a novel design comprising two non-colluding entities that together implement matching, while ensuring that neither entity learns the association between a user's ID and their location. This property is important since the matching service is a third-party as



(a) Architecture

Registering Items and Triggers
Item = CreateItem(Content, TTL)
Trig = CreateTrigger(Callback, TTL)
Create a new item or trigger.
Adding Attributes
AddLocAttr(ItemOrTrig, Loc, AutoUpdate)
AddAppAttr(ItemOrTrig, Attr)
Associate location or other attributes to items and trigger
Notifications
Callback.Notify(Content)
Called by platform when a match is found.

(b) API

Figure 1: Koi Platform

far as the user is concerned and so should not be in a position to learn the user's location. The third element is support for *rich, semantically-meaningful, multi-attribute matching* that arises from the observation that applications have diverse requirements. Indeed, location-based applications are about more than just location. For example, a user might want to know when grocery store is nearby, so whether a store qualifies as a "grocery store" is equally important as proximity of the user to the store. Further, what qualifies as a "grocery store" depends on the application; for example, one could choose to treat a store tagged as a "supermarket" also as a grocery store. Even within the narrow domain of the location attribute, what qualifies as "nearby" is app-dependent in a way that opaque location tokens cannot accommodate.

We have designed and implemented the Koi system, and have deployed the matching service running on a production cloud platform. We show through micro- and macrobenchmarks that our implementation of Koi is performant enough for practical use despite the overhead imposed by privacy constructs. As we elaborate on in Section 7, Koi's location-based matching API naturally accommodates applications such as social networking, local search, recommendations, and advertising, which are essentially based on matching. Finally, as discussed in Section 6, we have verified using the ProVerif theorem-prover [33] that Koi protocols preserve location-privacy.

2 Overview of Koi

Koi comprises two components, one which runs on the user's mobile device and the other in the cloud (Figure 1a).

The mobile component of Koi interfaces between applications and the cloud component. To applications, it exposes a simple API (Figure 1b), which allows registration and updating of *items* and *triggers*, and provides notification through a callbacks. An *item* is a statement of fact. It contains information about an entity such as a user, a business, a vehicle (e.g., bus), etc., in the form of one or more *attributes* of the item, including its location. The *location attribute* is special in that an application can set it to be updated automatically by Koi, for example, a user's location as they move. *Triggers* are similar to items except that these represent queries, specifically a request for a callback when a match is found.

The mobile component of Koi interfaces with the cloud component by communicating with it to register items and triggers, and to set and update their attributes. The Koi cloud service comprises two sub-components — the *matcher* and the *combiner*[1] — which are assumed to be non-colluding. In broad terms, the *matcher* knows about identities of users (and other items) and also their attributes (including location), but it does *not* know the association between the users and their location or other attributes. On the other hand, the *combiner* knows the association between (anonymized) users and (encrypted) locations (and other attributes), but it does not know the actual identities or the attribute values. A privacy-preserving protocol enables the matcher and the combiner to perform matching without either of them learning about the association between the users and their locations, or more generally between the identities of items and their attributes. At the same time, knowledge of the (plain-text) location and other attributes enables the matcher to perform rich, semantically-meaningful matching based, for example, on geocoding, location proximity or spelling correction.

Multiple applications can use a common Koi service. Attributes names are name-spaced to the application registering the item to avoid name collisions. Applications may inter-operate by registering triggers for another application's attributes. Multiple Koi services (we envision a handful) may operate independently.

[1]As discussed in Section 5, our design also accommodates multiple combiners.

3 Goals, Non-goals, and Assumptions

Having sketched an overview of Koi, we now discuss the specific goals of the system, some non-goals, and the assumptions made.

The goal of Koi is to provide location functionality to applications that need it while ensuring that no third party (i.e., entity other than the user and the application) is in a position to learn the association between a user's identity and their location (or other attributes).

It is *not* a goal of Koi to prevent a malicious application from leaking a user's location information[2]. A malicious application is one where the application-developer intentionally and deliberately violates privacy. Our position is that the fact that a user has chosen to run an application implies an implicit trust in the application's non-maliciousness, even if the application might be buggy or may include arbitrary third-party code not audited by the application developer. Applications where the developer does not intend to violate privacy, but has a bug, or includes a third-party library (such as an ad control) that leaks user information (with our without the app-developer's knowledge) are not considered malicious. Koi protects against this latter class.

We assume that the location or other attribute itself is not sensitive, rather it is the *linkage* between the user identity and the attribute that is sensitive and needs to be protected. We leave it to the application developer to decide which non-sensitive attributes to register with Koi, depending on what matching functionality is desired.

The matcher and the combiner are assumed to be non-colluding with each other[3]. Furthermore, we assume an honest-but-curious attacker model for each of the matcher and the combiner. This means while the internal functioning of each entity is beyond scrutiny (e.g., they could try to glean information from the messages that come their way), the external interface of each entity must be conformant. We believe that this assumption reflects the real-world situation, where a service such as Google or Facebook would be wary of the PR backlash that might result from any externally-visible non-compliant behavior (e.g., active attacks) attributable to them. For example, if the matcher were to create and register fake users in an attempt to get matched with and thereby learn the location or other attributes of a real user, it would run the risk of being exposed when the real user realizes that they have been matched with a fake user.

4 Design

We present first the Koi platform API exposed to apps on the phone, followed by the Koi cloud service. We present the detailed protocol description in Section 5.

[2]We present coping strategies for malicious apps in Section 10.

[3]We discuss practical disincentives to collusion in Section 10.

4.1 Platform API

The platform service model is similar to a database trigger. The app registers *item*s with the Koi phone-agent. Items may correspond to users or content (e.g., photos). The app associates *attribute*s with an item. Attributes may be locations, keywords, or arbitrary data. The app also registers *trigger*s. Triggers specify one or more attributes that must match. When an item matching the trigger is registered (by another user or app, or by the same app), the app registering the trigger is notified of the item through a callback. Figure 1b lists the Koi API.

The app specifies location attributes symbolically (e.g., loc:self, or within 1 block of loc:self). The Koi phone-agent internally replaces loc:self with the actual lat-long. The agent optionally automatically updates the lat-long if the user's location changes. This amortizes the cost of acquiring user location across multiple apps, and avoids having to wake each app up whenever the user moves. By never exposing lat-long data to the app, Koi minimizes the app accidentally leaking it to third-parties.

The app may associate arbitrary content with an item. A social networking app may, for instance, register the user's push-notification service handle so another user can contact this user. A citizen-journalism app may, for instance, register a photo or URL etc. The content may additionally be encrypted, for instance in the social networking app, only friends with the appropriate key may recover the push-notification handle.

4.2 Privacy-Preserving Matching Service

The operation of the Koi cloud service is best illustrated through an example. Consider the scenario in Table 1 where users Alice and Chuck register an item indicating they are tour-guides for Bangalore and Boston respectively. Bob registers a trigger looking for a tour-guide at his present location. The goal is to match Bob, who happens to be in Bangalore, with Alice. Note that Bob's phone-agent uses his lat-long without first geocoding it to Bangalore. For simplicity, we assume each of them register some unique user ID (as the item content, or the trigger callback) through which they can be reached. Our location-privacy goal is to prevent the cloud service from associating this user ID with the user's location.

At a high-level the Koi cloud service operates as follows. Each item or trigger is treated as a collection of rows — one for each attribute; Table 2a presents this *logical* view, i.e., it is never actually constructed or stored, and is shown here only to aid the description. Note a simple database approach that stores the user and attribute,

ITEM (AliceID)	TRIGGER (BobID)	ITEM (ChuckID)
· TourGuide	· TourGuide?	· TourGuide
· loc:Bangalore	· loc:12.58N,77.38E?	· loc:Boston

Table 1: Original Data

Content/Callback	Attribute	RegId	AttrId
AliceID	TourGuide	P	A
AliceID	loc:Bangalore	P	B
BobID	TourGuide?	Q	C
BobID	loc:12.58N,77.38E?	Q	D
ChuckID	TourGuide	R	E
ChuckID	loc:Boston	R	F

(a) Logical View

Content/Callback	RegId
AliceID	P
BobID	Q
ChuckID	R
Attribute	**AttrId**
TourGuide	A, E
TourGuide?	C
loc:Bangalore	B
loc:12.58N,77.38E?	D
loc:Boston	F

(b) Matcher Partition

RegId	AttrId
P	A
P	B
Q	C
Q	D
R	E
R	F

(c) Combiner Partition

Table 2: Actual partitions stored by the Matcher and Combiner

while sufficient for matching, does not satisfy the privacy constraint. To achieve its privacy goals, Koi first associates a (random) RegId with each registered item or trigger, and a (random) AttrId with each attribute in the registration. For example, in Table 2a the TourGuide attribute in Alice's and Chuck's registrations are assigned different AttrIds A and E respectively, and both rows corresponding to Alice's item are assigned RegId P. How the RegId and AttrId are picked (and by whom) is described later in the Koi protocol section.

As mentioned, Koi partitions the logical table above into two halves that are placed on two different (non-colluding) entities, neither of which is able to link a registration with attribute(s) associated with it. The *matcher* stores Table 2b, and a *combiner* stores Table 2c. In the example, the matcher cannot place Chuck in Boston since he does not know the association between RegId R and AttrId F. At the same time, the combiner, which knows the link between R and F, doesn't know which user or what attribute they correspond to.

Matching is initiated by Bob when he registers his trigger (Q). Given a RegId Q, the combiner first queries the matcher for an associated AttrId (say, C); the matcher responds with AttrIds A, E, which the combiner maps to RegId P, R respectively. The matcher can answer this query since in Table 2b C maps to the query TourGuide? and A, E map to the answer TourGuide. The combiner then queries the matcher for another AttrId (D) associated with the original RegId; the matcher responds with AttrId B. This is because the matcher, which runs in the cloud, can geocode the plain-text attribute (loc:12.58N,77.38E?) associated with D to loc:Bangalore?. Note the matcher can support arbitrary matching algorithms here without affecting privacy; this rich matching is a feature unique to Koi. The combiner maps the matcher's response B to RegId P. At this point the combiner notes that RegId P matches both trigger attributes in Q, while R matches only one (without knowing what any of those attributes are). The combiner then informs the matcher that RegId P and Q match, and the matcher invokes the callback for Q (BobID) with the content registered for P (AliceID). Note that the combiner's queries for C and D (from Bob's trigger Q) are mixed with other ongoing queries for other users' trigger registrations (or mixed with cover traffic generated by the combiner) so the Matcher doesn't learn which AttrIds contributed to a given match, or even which AttrIds correspond to a single registration.

5 Koi Protocol

In this section we describe the three Koi protocols: registration, matching, and combining. We describe first the honest-but-curious matcher and single-combiner case. We then relax the restrictions on the matcher, and extend to multiple combiners.

5.1 Registration

The goal of the registration protocol is to create necessary state in the matcher and combiner for matching items to triggers (Table 2 illustrates this state, and Table 3c lists the protocol).

R1. The client encrypts each attribute ($attr_j$) associated with the item or trigger first with the matcher's public-key (M), and then with the combiner's public-key (C). Probabilistic encryption (e.g., by adding randomized padding) is used to defend against dictionary attacks. The double-encrypted attributes are sent to the matcher along with arbitrary $user$ data, which is the item content or trigger callback handle. As mentioned, item content may be encrypted so only certain users can decrypt it.

R2. The matcher picks a random registration ID rid for the registration, and forwards the double-encrypted attributes along with the rid to the combiner. Note multiple items/triggers from the same user are assigned different (random) rids.

The matcher stores in a table R2U a mapping from the rid to the arbitrary $user$ data and the ID of the registering user U.

R3. Upon receiving the double-encrypted attributes, the combiner decrypts them to reveal j attributes for that registration encrypted by the matcher's public-key. The combiner picks a random attribute ID aid_j for each of the j encrypted attributes. It sends aid_j and the j^{th} encrypted attribute to the matcher (one at a time). These messages are mixed with aid/encrypted-attribute pairs from other ongoing registrations so the matcher can't link them back to which registration (rid) they are associated with. If enough natural cover traffic doesn't exist, the combiner can generate cover traffic.

The combiner stores the set of aid_j associated with the registration (rid) in the table `R2A`, and a reverse mapping from the aid_j to the rid in table `A2R`.

The matcher, upon receiving each aid/encrypted-attribute pair, decrypts the encrypted attribute to reveal the pain-text attribute $attr$. At this point the matcher may arbitrarily process the attribute to construct a set of equivalent $attr'_k$ For instance, the (misspelled) $attr$ Bretney may be corrected to include Britney in the equivalence set, or the location loc:12.58N,77.38E may be geocoded to include loc:Bangalore in the equivalence set. The matcher then updates table `T2A` for each $attr'$, which given a plain-text attribute returns the set of aids associated, to include the aid sent by the combiner. It also stores a reverse-mapping from aid to the plain-text set of equivalent attributes $attr'$ in table `A2T`.

This concludes the registration protocol.

5.2 Matching Attributes

Matching is event-driven: when a trigger (or item) is registered, the combiner looks for items (or triggers) matching it. Finding a match is relatively straight-forward (Table 3d). Given a registration (rid), the combiner looks up in table `R2A` the associated set of aid_j. It then executes the following two-message protocol individually for each aid_j.

M1. The combiner sends the aid_j to the matcher. As before, these messages are mixed with aids from other ongoing matches requests so the matcher can't link which sets of aids are associated with a single registration. And if enough natural cover traffic does not exist, the combiner can generate this cover traffic.

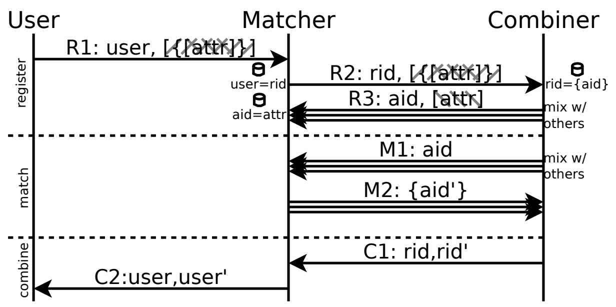
The matcher, upon receiving the message, looks up in table `A2T` the plain-text equivalent attribute set $\{attr'_k\}$ associated with that aid. For each $attr'$ in the equivalence set, the matcher looks up table `T2A` for the set of aid's registered. The matcher unions together these sets of aid's to construct the response.

M2. The matcher sends the response back to the combiner.

For each aid', the combiner queries table `A2R` to retrieve the registration rid' associated with that aid'. The combiner marks rid' as a match for the j^{th} attribute for

$[m]_X$	Probabilistic encryption of m using public-key of X
U, U', M, C —	User, User 2, Matcher, Combiner
$user$	Arbitrary user data for item content; or callback key for trigger
$attr$	Attribute (plain-text)
rid, aid	Registration ID, Attribute ID
`R2U`, `R2A`, `A2R`, `T2A`, `A2T`$[x] \leftrightarrow y$ —	Table $key : x, val : y$; store ($\longleftarrow$), lookup ($\longrightarrow$)

(a) Notation



(b) Protocol. Encrypted parts hatched; Matcher key: \, Combiner key: /

#	From → To Message	Processing
R1.	$U \rightarrow M$ $user, [\{[attr_j]_M \vert \forall_j\}]_C$	
R2.	$M \rightarrow C$ $rid, [\{[attr_j]_M \vert \forall_j\}]_C$	**Matcher:** `R2U`$[rid] \leftarrow \langle U, user \rangle$
Repeat $\forall_j$ mixed with messages from other ongoing registrations.		
R3.	$C \rightarrow M$ $aid_j, [attr_j]_M$	**Combiner C:** `R2A`$[rid] \leftarrow$ `R2A`$[rid] \cup \{aid_j\}$ `A2R`$[aid_j] \leftarrow rid$ **Matcher** $\{attr'_k\} \leftarrow process(attr_j)$ $\forall_k$`T2A`$[attr'_k] \leftarrow$ `T2A`$[attr'_k] \cup \{aid_j\}$ `A2T`$[aid_j] \leftarrow \{attr'_k\}$

(c) Registration

#	From → To	Message	Processing
			Combiner C `R2A`$[rid] \rightarrow \{aid_j \vert \forall_j\}$
Repeat $\forall_j$ mixed with messages from other ongoing match requests.			
M1.	$C \rightarrow M$	aid_j	**Matcher** `A2T`$[aid_j] \rightarrow \{attr'_k\}$ $\bigcup_{\forall_k}$ `T2A`$[attr'_k] \rightarrow \{aid'_l \vert \forall_l\}$
M2.	$M \rightarrow C$	$\{aid'_l \vert \forall_l\}$	**Combiner C, $\forall_l$** `A2R`$[aid'_l] \rightarrow rid'_l$ $\mathcal{M}_{rid}[j] \leftarrow \mathcal{M}_{rid}[j] \cup rid'_l$

(d) Matching

#	From → To	Message	Processing
For each $rid' \vert \forall_j rid' \in \mathcal{M}_{rid}[j]$			
C1.	$C \rightarrow M$	rid, rid'	**Matcher** `R2U`$[rid] \rightarrow \langle U, user \rangle$ `R2U`$[rid'] \rightarrow \langle U', user' \rangle$
C2.	$M \rightarrow U$	$user', user$	

(e) Combining

Table 3: Koi Protocol Description

the rid for which the matching protocol was initiated (in match-set $\mathcal{M}_{rid}$).

Note, by design, there are no cryptographic operations in the match protocol to support high matching throughput.

5.3 Combining Matches

The simplest criteria the combiner can use to determine if rid' matches rid is if rid' is present in the match-set $\mathcal{M}_{rid}[j]$ for all j. The combiner can support richer criteria (e.g., boolean match expressions), which we omit for brevity.

C1. Once a match is found, triggering the callback is straightforward. The combiner notifies the matcher that registrations rid, rid' match. The matcher looks up in table `R2U` the associated users and content/callback data. We discuss below a slight modification of this that hides which pair of rids matched from the matcher.

C2. The matcher sends to the user registering the trigger, say user U, the $user$ data for the callback handle that fired, and the $user'$ data for matched item content.

User U's phone-agent, upon receiving notification C2, invokes the app-registered callback with the matched item content. The app is free, at this point, to present the content to the user (e.g., if the content is a photo), or initiate direct communication with the other user (e.g., if the content is a push-notification ID for the other user). The action the app takes in the callback function are external to Koi.

5.4 Extensions

We now discuss two extensions that further reduce the information learned by the matcher without affecting functionality.

Keeping the matched pair secret. In message C1 above, although the matcher doesn't learn which location-attribute contributed to the match, the matcher still learns which pair of users matched (which might implicitly leak some information). To exchange $user$ data without either the combiner or matcher learning what was exchanged and with whom, and doing so without establishing shared keys between the matched users, we use a commutative cryptography scheme [41]. In commutative crypto, a message encrypted first by key k_1 and then by key k_2 can be decrypted using the (inner) key k_1 to reveal the message singly encrypted by only the (outer) key k_2.

To keep the matching secret from the matcher, in message R1, the user encrypts the $user$ data with a per-registration randomly chosen key k_{rid} using a commutative encryption scheme; it sends k_{rid} to the combiner by putting it inside the R1 message component encrypted with the combiner's public-key (which the matcher forwards to the combiner in R2). The matcher additionally encrypts the encrypted $user$ data with a secret key (k_M), and includes the double-encrypted $user$ data in message R2. The combiner retrieves k_{rid} from the message and uses it to decrypt the double-encrypted $user$ data. By the commutative encryption property, the combiner now has the $user$ data for each rid encrypted with the matcher's secret key.

When notifying the user of the match, instead of sending message C1, the combiner retrieves the (encrypted) $user'$ data associated with rid', encrypts it using the k_{rid} registered by user U, and sends it to the matcher directing it to forward to the user for rid. The matcher decrypts the double-encrypted data using his secret key k_m, revealing $user'$ single encrypted with k_{rid}, which it sends to user U who can decrypt it to reveal the $user'$ data registered by the matched user.

During the protocol neither the matcher nor combiner learn the content of $user$ data, and during the notification phase encryption prevents the matcher from learning which other user's $user'$ data was sent to user U.

Multiple combiners. It may be tempting for the matcher to collude with the combiner if there is only one combiner. Having many combiners allows the user to pick which combiner he trusts to not collude with the matcher. Supporting multiple combiners requires a small change to the protocol. The user, in message R1, indicated to the matcher his choice of combiner C_x and uses C_x's public-key to encrypt the second component of the message. The matcher then forwards the message to C_x in R2. Random aids chosen by the combiner are namespaced to the combiner, e.g., by prefixing the combiner's domain name to the aid.

During matching, in message M2 a combiner receives aids registered both by itself and other combiners. For aids the combiner itself registered (which it can tell by the namespace prefix), it follows the protocol as before. For aids registered by other combiners, the combiner constructs all possible sets of the form $\{aid'_j | aid'_j \in \mathcal{M}_{rid}[j]\}$ where the aid' all belong to combiner C_y; each set corresponds to a *possible* rid' registered with C_y because the combiner doesn't know which aid's belong to a single registration, vs. which are from different registrations. It then engages in a private set intersection protocol [15] with C_y to determine if any of the aid' sets matches an actual registration; if so, the two combiners exchange the encrypted $user$ data (above) and notify the users of the match. If none of the aid' sets match, by the properties of the private set intersection protocol, neither combiner learns anything in the process.

6 Privacy Analysis

In this section we first define informally what we mean by location-privacy and our trust assumptions. We then model Koi in applied pi-calculus [35], a language for formally modeling distributed systems and their interactions, which then allows us to use the ProVerif [33] automated cryptographic protocol verifier tool to provide machine-generated proofs of Koi's privacy properties. We then discuss privacy concerns *external* to the Koi service, arising from poor application design, and offer coping strategies for applications.

6.1 Defining Privacy

Our privacy goals are based on Pfitzmann and Köhntopp's definition of anonymity [31] which is unlinkability of an *item of interest* (IOI) and some logical user identifier. Pfitzmann and Köhntopp consider anonymity in terms of an *anonymity set*, which is the set of users that share the given item of interest — the larger this set, the "better" the anonymity. In the related-work section (Section 11) we compare this definition of privacy to k-anonymity, l-diversity, and differential privacy.

In the Koi context, anonymity translates to unlinkability between an attribute ($attr$) and the registration ID (rid) (and by extension, the registering user U) the attribute is associated with. We assume that the individual attributes themselves are *not* sensitive; rather, it is when these attributes are *linked* to the user, or to the user's other attributes narrowing down and possibly identifying the user, that it becomes sensitive. Thus for location-privacy, as long as location data is present only in item/trigger attributes, it cannot be linked back to the user or his other attributes, thereby preventing the user from being tracked by third-parties.

6.2 Proving Privacy Properties

Before we model Koi, we recall basic ideas and concepts of applied pi-calculus needed for our analysis. A more comprehensive description (in the ProVerif context) is available in [9].

6.2.1 Applied Pi-Calculus Primer

Messages. Messages are obtained by applying *constructors* on *names*, *variables*, and other messages. Constructors are function symbols e.g., $\mathsf{REncrypt}(\dots)$. Names are symbols for atomic data e.g., Alice. Variables e.g., x, may be bound to names or messages. Messages are taken apart by *destructors*. Destructors are function symbols e.g., $\mathsf{RDecrypt}(\dots) \rightarrow \dots$.

Equations of the form $x = y$ can be used to establish the equivalence of two messages.

As illustration, we model probabilistic asymmetric encryption as the following pi-calculus relation:

$$\mathsf{RDecrypt}\left(\mathsf{REncrypt}(m, \mathsf{PubKey}(k), r), k\right) \quad \rightarrow \quad m \tag{1}$$

In Equation 1, the constructor for probabilistic encryption $\mathsf{REncrypt}(m, pk, r)$, the r component is fresh for every encryption thus preventing dictionary attacks. The destructor $\mathsf{RDecrypt}(\dots, k) \rightarrow m$ succeeds only if pk (from the constructor) and k are related in the form $pk = \mathsf{PubKey}(k)$. If so, the destructor discards r and yields the message m.

Channels. Channels are named sources/sinks for messages. A message m sent to channel c using $\mathtt{out}(c, m)$ can be received using $\mathtt{in}(c, x)$, where x will be bound to m. Channels model asynchronous out-of-order message exchange.

Processes. Processes are built from the grammar below. We omit discussion of the syntax (see [33] for details).

P,Q,R :=	processes
0	null process
P | Q	parallel composition
!P	replication
new a; P	name restriction
let N = D in P else Q	term evaluation
if N = M then P else Q	conditional
in(c,N); P	message input
out(c,N); P	message output

As illustration, we model a simplified version of the matcher fragment that processes message R1 and generates message R2 (from Table 3) as follows:

```
MatcherR1R2 :=
    in(net, m);
    let (user, eattrs) = m in
    let rid = RID(m) in
    out(net, (rid, eattrs))

Matcher := ... | !MatcherR1R2 | ...
```

The matcher is modeled above as the parallel combination of smaller processes that each process one type of message. The process MatcherR1R2 receives a message m from the channel net; deconstructs m to extract the $user$ data, and the encrypted attribute $eattrs$; picks a fresh registration ID for the message m using the RID constructor; and sends out the message $(rid, eattrs)$ back on the net channel. In our full Koi model, messages are tagged with a message type to avoid ambiguity in processing.

6.2.2 ProVerif Primer

ProVerif [33] can verify, among other security properties, the secrecy of information in protocols expressed in the applied pi-calculus in a fully automated manner. Applied pi-calculus models distributed protocols as a collection of messages, parallel processes, and channels.

ProVerif is sound. ProVerif performs a brute-force exploration through the proof space. When it says that a security property is true, then it actually is so. The security proofs obtained through ProVerif are valid for an unbounded number of sessions of the protocol. However, ProVerif is not complete (i.e., false attacks can be found). When ProVerif finds an attack it provides a derivation tree that can be used to manually verify whether the attack found is false or not.

ProVerif is *not* always the preferred choice for verifying new cryptographic constructions because it assumes perfect cryptography, but is useful in verifying straightforward uses of existing cryptographic primitives, as is the case with Koi. This is because if an attack were to be found against a specific instantiation of a crypto primitive (e.g., RSA), its use in Koi is trivially replaced with another instantiation (e.g., ElGamal).

ProVerif's default attacker model models a Dolev-Yao attacker [11] that can overhear, intercept, and synthesize any message sent on any channel it has access to. This is too weak since the attacker doesn't have access to internal state of the participants. ProVerif allows for more powerful attacker models by allowing the user to specify what internal state the attacker can access or modify.

Proverif cannot model traffic analysis attacks. As mentioned, we assume there is enough natural (or generated) cover traffic for creating a mix [5] that mixes, delays and reorders messages to defeat traffic analysis and timing attacks.

6.2.3 Modeling Koi

We have modelled Koi in applied pi-calculus. Our model includes the extension that keeps matched pairs secret from the matcher. The model was constructed by one of the authors, and then manually and independently checked by the other two authors. One of the authors checking the model had experience with implementing Koi, and used this knowledge to verify correctness. The other author checking the model used only Section 5 of this paper to cross-verify the applied pi-calculus model and the protocol description here. As a final sanity-check, we introduced a number of bugs in the protocol (e.g., using deterministic encryption, not encrypting something that should be, etc.) and ensured that ProVerif found attacks that exploited the bugs introduced.

We highlight below some non-trivial aspects of our model, which we view as methodological contributions of our work.

Unlinkability. ProVerif does not natively model unlinkability. That is, if process u_1 were to send out message m_1, and process u_2 were to send out m_2, ProVerif tracks only that the attacker knows all four names (u_1, u_2, m_1, m_2) but does not track the *link* between u_1 and m_1 or u_2 and m_2.

We model unlinkability by creating a new constructor $\mathsf{LINK}(x, y)$ that explicitly models the linkability of x and y, and a new destructor INFER that allows the attacker to infer new links as follows.

$$\mathsf{LINK}(a, b) = \mathsf{LINK}(b, a) \quad (2)$$

$$\mathsf{INFER}(\mathsf{LINK}(x, a), \mathsf{LINK}(a, y)) = \mathsf{LINK}(x, y) \quad (3)$$

Equation 2 states that links are symmetric, i.e, if a is linked to b, then b is linked to a. Equation 3 states that links are transitive, i.e, if x is linked to a, and a is linked to y, then x is linked to y.

For each message sent on a channel we emit LINK messages that the attacker may use to draw inferences and new conclusions from. Thus unlinkability of x, y translates to ProVerif's notion of secrecy of $\mathsf{LINK}(x, y)$, which ProVerif is well-equipped to prove in an automated manner.

Adversary. The standard ProVerif attacker can observe only messages sent on public channels, and cannot observe internal state (e.g. if a process decrypts a message and then re-encrypts it before sending it out on a channel, the attacker would not have access to the decrypted message). To model scenarios where the process itself may be compromised by an attacker (since in our model the matcher and combiner could themselves be adversaries), we use the notion of a "spy" channel on which all internal internal process state is echoed (e.g., the decrypted message above) as well as give the spy write access to all channels the process has access to, in effect allowing the ProVerif attacker to completely supplant the compromised process. We model a spy channel for each Koi entity. By giving the ProVerif attacker access to the appropriate spy channel we can model an adversarial matcher, or combiner. By giving the attacker access to multiple spy channels we can model collusion between adversarial parties. By setting the ProVerif attacker to passive we model honest-but-curious adversaries (i.e., can receive but not send messages).

Datastore. ProVerif's constructors/destructors do not modify the environment and therefore cannot be used to model stateful operations such as a datastore. The only place to "store state" is in an (asynchronous) channel, where a message is "stored" between when it is sent and when it is received. We model storing into a datastore $\mathcal{M}[x] \leftarrow y$ as $\texttt{out}(\mathcal{M}, (x, y))$. We model performing a lookup on x as the sequence $\texttt{in}(\mathcal{M}, (= x, y))$; $\texttt{out}(\mathcal{M}, (x, y))$. The $(= x, y)$ syntax is ProVerif syntactic sugar that uses pattern-matching to deconstruct the input message into a tuple, check that the first element equals x, and binds the variable y to the second element. Since `in` removes the message from the channel (the equivalent of deleting the mapping from the datastore each time a lookup is performed), after each lookup we add back the mapping into the datastore (by using `out`). Giving the attacker access to the appropriate datastore channel is an easy way to model an adversary that has read/write access to this internal state.

6.2.4 ProVerif Results

We model a configuration with one (honest) user U_1 — the intended victim of privacy violations — and an unbounded number of other users, a matcher, and a combiner. The honest user registers two attributes A_1 and A_2. We ensure at least one other user (U_2; honest or not) registers A_1 so he is matched with U_1.

We then ask ProVerif whether the attacker can conclude the following under various assumptions regarding which parties are being adversarial.

P1. $\mathsf{LINK}(U_1, A_1)$, i.e., U_1 registered A_1
P2. $\mathsf{LINK}(A_1, A_2)$, i.e., there exists some user that registered both A_1 and A_2
P3. $\mathsf{LINK}(U_1, U_2)$, i.e., U_1 was matched with U_2.

Using ProVerif we were able to generate proofs for the following privacy properties:

Result 1: A honest-but-curious combiner cannot violate P1, P2, or P3. This is easy to see since the combiner is never sent a message that contains user ID, $user$ data, or attributes in an unencrypted form.

Result 2: A honest-but-curious matcher cannot violate P1, P2, or P3. As with all properties proved by Proverif, this holds for an unbounded number of messages in arbitrary order, subject to the mixing and crypto assumption mentioned earlier.

Result 3: A honest-but-curious combiner in collusion with a honest-but-curious matcher can violate P1, P2 and P3. As expected, if both the matcher and the combiner collude, the Koi service, as a whole, is analogous to existing cloud services that are organized as a monolithic database. Such a database can trivially violate all privacy properties above since it would have full knowledge. Thus as long as the combiner and matcher do not collude, our privacy goals P1, P2 and P3 are assured.

7 Koi Use-Cases

We examined 10 most popular location-based applications on the Android platform. The functionality provided by these applications can broadly be classified into 6 classes as listed in Table 4. We describe in pseudocode example applications we have built for two of these classes, and discuss briefly the other classes of apps.

Application	SN	TC	LS	LR	LA	ND
BrightKite	✓	✓	✓	✓		
Facebook Places	✓	✓	✓	✓		
Foursquare	✓	✓	✓	✓		
Google Latitude	✓	✓				
Google Maps						✓
Google Search			✓			
Google Ads					✓	
Gowalla	✓	✓	✓	✓		
Loopt	✓	✓	✓	✓	✓	
Routes	✓	✓				

Table 4: Location-based functionality provided by popular applications. SN: Social Networking, TC: Tagging Content, LS: Local Search, LR: Local Recommendations, LA: Location-based Advertising, ND: Navigation Directions.

7.1 Private Mobile Social Network

A mobile social network facilitates interaction between nearby friends and friends-of-friends. With the existing instantiations (e.g., Foursquare or Facebook Places), the need to perform matching on *friends* who are *nearby* violates privacy in two significant ways: first, the cloud service learns the identities of a user's friends; and second, the service also learns each user's location.

We have implemented a novel mobile social networking application on Koi, which enables nearby friends and friends-of-friends with common interests to get in touch, in a privacy-preserving manner. The user's profile is hidden from the OSN service, which also means that a user's profile can only be seen by others whom the user allows.

Here is how the application operates. The user creates his profile on his phone by running the application. He adds to his profile information including his interests, photos, etc. He also picks a random key that others must possess before the application will grant them access to data in his profile. He adds friends by simply exchanging profile keys with them. The application registers an item for the user (Algorithm 1, line 2– 8) with the user's push-notification mailbox as the item content, an encrypted attribute with the user's name prefixed by me:, an (auto-updating) attribute with the user's location, and an encrypted attribute for each of his friends' names prefixed by friend:. The app then registers a trigger for each friend (line 9–13). One attribute identifies the friend of interest (line 12), which matches the name registered by the friend's app on line 4, and another targeting 1 mile from the user's current (auto-updating) location (line 13), which matches the location registered by the friend's app on line 5. As the user and his friends move around, their phone-agents update the location attributes. When a friend moves within 1 mile of the user, the corresponding trigger fires with the friend's push-notification mailbox, which the app can use to exchange messages.

For matching friend of friends, the me: on line 11

```
 1: procedure MatchFriends(User)
 2:     I ← CreateItem(User.PushMailbox, ttl)
 3:     U ← Encrypt(me: + User.Name, User.ProfileKey)
 4:     AddAppAttr(I, U)
 5:     AddLocAttr(I, loc:self, true)
 6:     for all F in User.Friends do
 7:         P ← Encrypt(friend: + F.Name, F.ProfileKey)
 8:         AddAppAttr(I, P)
 9:     for all F in User.Friends do
10:         T ← CreateTrigger(F.OnNear, ttl)
11:         V ← Encrypt(me: + F.Name, F.ProfileKey)
12:         AddAppAttr(T, V)
13:         AddLocAttr(T, loc:self+1 mile, true)
```

Algorithm 1: Private Mobile Social Network Application

```
 1: procedure Route(Dest)
 2:     I ← CreateItem(Null)
 3:     AddLocAttr(I, route.direction:⟨loc:self;Dest⟩, true)
 4:     N ← RandomNonce()
 5:     AddAppAttr(I, N)
 6:     for all act in Actions do
 7:         T ← CreateTrigger(act.Announce)
 8:         AddAppAttr(T, N)
 9:         AddAppAttr(T, act)
```

Algorithm 2: Turn-by-turn Directions Application

is changed to read friend:, which then matches the attribute for the common-friend registered by the friend-of-friend on line 8. Matches can be restricted based on shared-interests by associating attributes with the user's item, and querying for them in the triggers. Groups (e.g. photography-club) is supported in an identical manner with the group's name and group's secret key used instead of the friend-name and friend's key.

Note that neither the cloud, nor non-friends can track the user. As per Koi's location-privacy guarantees, the user's location cannot be tracker by the cloud service. A stranger cannot track the user's location since matching the user's item requires the stranger to construct a trigger with the encrypted username attribute (line 11), which only those that have the user's profile key can construct.

7.2 Turn-by-turn Directions

We created a proof-of-existence turn-by-turn directions application to demonstrate how Koi can add privacy to navigation applications that make heavy use of location information and cloud knowledge. Today any query for driving directions reveals to the cloud third-party (typically Google, Bing, or MapQuest) the user's current location and intended destination. Koi avoids both.

A key challenge in building a turn-by-turn direction application using Koi is that the Koi APIs do *not* provide a way to directly query for information such as the route to the destination. To get around this problem, we leverage that fact that the matcher sees attributes in clear-text and could employ application-specific logic while performing matching. So the end-to-end route is decomposed to waypoints, which are then be conveyed to the application one-by-one, via triggers registered with Koi.

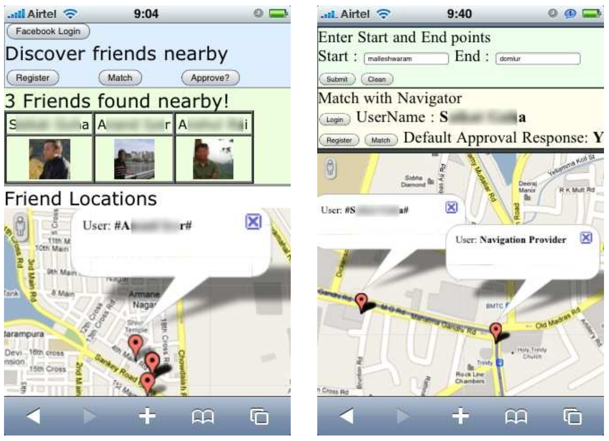
In particular, the application registers an item with an attribute that indicates the user's current location and the destination (Algorithm 2, line 3). This item implicitly corresponds to the next waypoint on the end-to-end route. The application also anticipates and registers one trigger for each possible directive corresponding to the next waypoint (e.g., "go straight", "turn left", "turn right", etc.; lines 2.6–2.9). Based on the current location and destination contained in the attribute, the application-specific matcher logic looks up the route and replaces the item's attribute with the actual directive corresponding to the next (i.e., first) waypoint. (Such a replacement is akin to the matcher applying spelling correction or using synonym information as part of its application-specific matching logic.) Say the directive at the first waypoint is "turn left"; the trigger corresponding to "turn left" fires and so the application learns that the next directive to present to the user is "turn left". As the user travels along the route, the phone-agent auto-updates the user's location, which causes the matcher to compute the direction to the next waypoint, which fires the trigger corresponding to the next directive, and so on. In this manner, the turn-by-turn directions are conveyed to the user.

7.3 Local Search, Tagging, Advertising

The four remaining classes of apps identified in Table 4, i.e., location-based content tagging, local search and recommendations, and location-based advertising, essentially all boil down to registering an item with the appropriate content and attaching location and other attributes to it. The item can be found by others using the appropriate triggers.

8 Implementation

We have implemented the Koi platform and the two proof-of-concept applications mentioned. The cloud-component is written in 1040 lines of C# code. The phone-based agent is available as a 230 line Javascript library that can be used by HTML5 applications on all modern smartphones, and as a C# library that can be used by native applications on the WP7 platform. The C# agent can leverage the platform's native push-notification service to receive trigger updates, while the Javascript version resorts to polling. The cloud service exposes an open REST-based API (over HTTP), allowing agents to be written for other platform and programming languages.


Airtel 9:04
Facebook Login
Discover friends nearby
Register
Match
Approve?
3 Friends found nearby!
S[redacted]a A[redacted]r A[redacted]i
Friend Locations
User: #A[redacted]r#
Airtel 9:40
Enter Start and End points
Start : malleshwaram End : domlur
Submit Clean
Match with Navigator
Login UserName : S[redacted]a
Register Match Default Approval Response: Y
User: #S[redacted]a#
User: Navigation Provider

(**a**) Mobile Social Network (**b**) Navigation App

Figure 2: Two applications implemented on the Koi platform

In addition to the core Koi API, our agent exposes a GUI API since location-based apps naturally use map widgets that require lat-long information. Our GUI API is a bare-boned API to allow the app to put push-pins at the user's current location (specified symbolically as loc:self), and overlay simple geometric shapes. Figures 2a and 2b show screenshots of early prototypes of our proof-of-concept applications. These applications consist of 50–60 lines of Javascript code, and required around 6 person-hours each to create.

9 Experimental Evaluation

We view a significant portion of the contribution of this paper as being architectural, in terms of proposing a higher-level abstraction (location-based triggers) as an alternative to the lat-long location API that is commonly used. As such, the evaluation of this architectural contribution rests on showing the ease of building applications on the Koi platform, which we have done to a small extent in Section 8. In this section we focus on the performance of the Koi cloud service, which we evaluate experimentally using both real-world traces, and micro-benchmarks. In order to exclude evaluation artifacts arising from load on the Azure platform, all experiments are run on one core of a 3 GHz dual-core machine with 4 GB of memory; the matcher and combiner share the one core while the second core is used by the benchmarking process. We use the loopback device for all communications to test the raw performance of our implementation independent of network bandwidth and latency.

9.1 Macro-Benchmark: Mobile Ads

We benchmark a location-based advertising application where business-owners register advertisements with location attributes, and users are matched to ads for businesses near them (i.e., within a hundred meters). We use a 2 GB real-world mobility trace of 264K users from around the world collected over a period of 1 year. The trace contains 22 million timestamped latitude-longitude updates across all users. Since we lack advertiser information, we simulate 10K businesses near popular locations visited by many users.

Processing 12 months worth of trace data through our implementation takes around 2 hours and 50 minutes. The system generated 2.2 billion notifications, all within 30ms of the location update that triggered the notification. The peak memory consumption is around 2.5 GB over the entire run. Even considering the small size of our trace we find our implementation running on a single-core can easily handle a mobile advertising application while leaving plenty of headroom for more demanding applications.

That said, macro-benchmarks, even for a handful of applications, are by nature inadequate for evaluating performance limits. We turn to micro-benchmarks to stress our implementation to its limits.

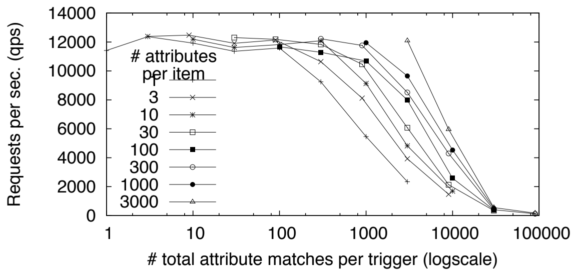


Figure 3: User matching performance

9.2 Micro-Benchmarks

Matching. We focus first on the matching throughput of our Koi implementation since matching is the primary mode of use. Our implementation combines messages R3 and M1 (Table 3) into a single message to amortize communication costs. Figure 3 plots the number of matching queries processed successfully per second (qps) as a function of how many attributes matched. That is, if the trigger matched 100 items each on 10 attributes, or if the trigger matched 1000 items on 1 attribute each, the x-axis value for both datapoints is 1000. The y-axis value is the mean query throughput for processing 100K requests (i.e., each datapoint represents an experiment lasting between 10s to a few minutes); standard-deviation error bars are negligible.

As is evident from Figure 3, end-to-end performance saturates to its peak as long as the average number of matching attributes (per request) is below 100. The bot-

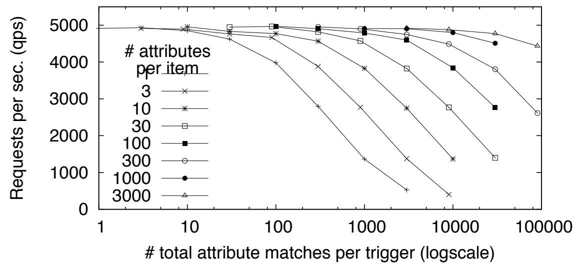


Figure 4: Attribute registration (and indexing) performance

tleneck here is the connection throughput of the HTTP library (which saturates are 12K qps.) Between 100 and 10K matching attributes, performance depends on the number of items matched (with fewer items resulting in higher performance); this is because our in-memory index layout is optimized for triggers matching a few items on many attributes. Beyond 50K attributes per match performance plummets as we run into memory issues. To put this in perspective, for the macro-benchmark above, this quantity is in the low tens, representing several orders of magnitude of headroom.

Registration. We focus next on registration throughput which includes both cryptographic operations, and building the various tables and indexes (e.g., `R2U`, `R2A`, `A2R` etc.) Figure 4 plots the number of registration requests processed per second (qps) as a function of the average number of matching attributes per trigger registered (which governs our in-memory index build process). As above, datapoints represent mean qps for processing 100K requests.

The figure is qualitatively similar to Figure 3 in that performance peaks for workloads similar to the advertising benchmark. The peak performance in Figure 4 is half that of Figure 3 since twice the number of HTTP requests are involved and the bottleneck is the HTTP connection throughput. As before, performance degrades as our index runs into memory issues, though more gradually due to different access patterns.

Combining. Our implementation can process trigger notifications at the rate of 12K qps (same peak as Figure 3). This throughput is independent of the number of items and attributes.

Overall we find our prototype implementation performs well enough even for real-time apps, and moves the performance bottleneck outside of Koi and into the communication layer.

10 Discussion

We discuss briefly privacy-related issues beyond Koi.

The first issue pertains to malicious applications registering a large number of finely-spaced triggers or triggers at sensitive locations, and reverse-engineering a victim user's location based on triggers matched. A weak defense against this attack would be for Koi to rate-limit the number of trigger registrations from an application (either per-device, or across all devices). This would force the attacker to Sybil himself to remain below threshold. Distributing apps through mobile marketplaces today requires developers to purchase a developer key. If the Sybils were to re-use this key, or re-use the credit-card used to purchase this key, they would be easily linked. Rate-limiting combined with an economic burden could serve to proactively mitigate against malicious applications.

The second issue pertains to collusion between the matcher and combiner. We mitigate the possibility from both the matcher and combiner side. On the combiner side, we allow for privacy-advocacy firms (e.g., EFF and ACLU), anti-virus companies (e.g., McAffee), certificate agencies (e.g., Verisign), non-profits (e.g., Mozilla), and other such outfits to run the combiner. Since the existence of these outfits is entirely dependent on public trust, it creates a strong disincentives to collusion — if they collude and anyone finds out (e.g., during an audit, or through whistle-blowers), the company would lose all credibility and be forced to shut down (as happed with the DigiNotar certificate agency recently). On the matcher side, by allowing the user to pick the combiner (of which there may be hundreds) we make it infeasible for the matcher to collude with any significant fraction of these combiners before it becomes public. The matcher would then be guilty of intentionally and deliberately circumventing privacy technology, which they could be legally liable for, creating a strong disincentive.

The third concern pertains to incentives for apps to adopt Koi. At a high level, it is up to the platform to drive adoption. Incentives may include positive reinforcement (e.g., higher placement in the mobile marketplace for Koi-enabled apps), negative feedback (e.g., more frequent nagging popups for apps using legacy location APIs instead of the Koi API), or strong enforcement (e.g., blocking the legacy location API for free applications). Overall, the platform can use a combination of these and other incentives to drive adoption.

11 Related Work

Existing Location APIs in Production Use. As mentioned, Apple's iOS Core Location [1], Android's Location Manager, and the Windows Phone 7 Location Service all expose only low-level lat-long information to apps which, when carelessly passed by apps to third-party code (e.g., Google's AdMob or Apple's iAds widget), poses a privacy risk as reported in TaintDroid [14] where the third-party service can track the user across multiple applications.

Location APIs in Prior Research. There is a rich body of research on location APIs. For programming ease, Brown et al. [4] propose the notion of a stick-e note that applications can register, which is triggered whenever the user's present context (e.g., location) matches that specified in the note. Other APIs have focused on fusing location information from multiple sensors to provide applications a unified view [28]. The Location Stack [21] proposes a 7-layer model that combines sensing and fusion with the inference of user activity and intentions.

While the above work focuses on important issues such as programming ease, sensor fusion, and activity inference, we view these as complementary to our focus on privacy in Koi.

Location Privacy. There is a rich body of work on location privacy. A survey some of this work appears in [17] and some challenges are discussed in [2,36,38].

Myles et al. [30] propose a rich policy API that includes support for symbolic locations as well as geographic locations, and for triggered callbacks. However, their proposal depends on a trusted external entity to enforce privacy constraints. In contrast, PlaceLab [24] takes a decentralized approach, wherein end devices acquire location information locally and allows users control over whether and how to share this information. As an extreme form of decentralization, Anonysense [8] ensures that the mobile nodes in a participatory sensing context do *not* share their location at all. Instead, the nodes download all tasks registered with the system and then determine locally which ones match their location and should be executed. Besides imposing a bandwidth cost, such an approach is not suitable for location-based applications that depend on the location of other users, e.g., mobile social networking as in Four Square.

There is a large body of work on techniques to cloaking user location while still enabling location-based services. Gruteser et al. [20] propose a centralized location broker that performs temporal and spatial cloaking of user location information, keeping in mind such factors as the density of users in a given area. In the context of a traffic monitoring application, the authors in [22] propose having virtual trip lines to trigger the reporting of location updates by mobile nodes, say based on the privacy sensitivity of locations. All of these techniques assume a trusted intermediary.

The research that is perhaps closest to ours in spirit is the recent work of Jaiswal and Nandi [26]. As in our work, they steer away from having a trusted intermediary by having multiple distinct entities, each holding a part of the sensitive information, work in unison to provide location-based services. However, location updates can still be linked, which opens up the possibility of attacks.

In comparison to the above work, Koi is designed explicitly to ensure privacy even with respect to the cloud service, which is not trusted. Also, unlike prior work, we do not seek to anonymize users or hide their location information. Rather, we consider the *linkage* between a user and their location as the privacy-sensitive information and focus on protecting this from the cloud service.

Notions of Privacy. Several notions of privacy have been proposed in the databases literature. k-anonymity [40] ensures that each output row is identical to at least $k-1$ other rows. However, k-anonymity is susceptible to attacks that exploit the lack of diversity in the value of sensitive attributes amongst the k rows. To address this, l-diversity [29] ensures that the output contains l well-represented values for each sensitive attribute. k-anonymity and l-diversity are complementary to Koi in that the combiner could suppress matches when the anonymity set size drops below k (i.e., fewer than k items match); the downside, however, is reduced functionality since matches cannot be too specific (e.g., cannot match nearby friend if only one friend is nearby).

Differential privacy [13] adds noise to ensure the output (typically aggregate queries) is independent of the presence or absence of a particular record. The differential privacy model is fundamentally a bad fit for Koi since the output of a matching service cannot be independent of whether the matched item is present or absent.

Privacy in Publish-Subscribe Systems. A publish-subscribe (pub-sub) system comprises publishers who post events, subscribers who register filters corresponding to events of interest to them, and a broker who *matches* events from publishers to the filters registered by the subscribers. Traditionally, the broker is assumed to be trusted, however, there has been recent work on the issue of confidentiality. Rich matching, while supporting confidentiality, is particularly hard for existing systems. [27] encrypts sensitive fields, but matching is restricted to the unencrypted fields. [37] employs a commutative encryption scheme [32] for confidentiality, however, matching is restricted to equality on a single keyword. [34] encrypts events and filters and then uses technique for search on encrypted data [12] to match these, but matching is limited to equality matching, keyword matching, and some limited numeric range matching. PSGuard [39] encrypt using hierarchical key derivation algorithms [42]; such key spaces are constructed for different types of matching, including topic or keyword matching, numeric attribute based matching, and prefix or suffix matching. Finally, [25] uses attribute-based encryption (CP-ABE) [3] to encrypt the events and key-policy attribute-based encryption (KP-ABE) [18] to encrypt the filters, and combine KP-ABE with searchable data encryption [12] to enable the broker to perform matching that can be expressed as conjunctions and disjunctions of equalities, inequalities and negations. Koi supports rich matching that goes far beyond the functionality of prior schemes by enabling matching based on application-specific semantics (e.g., matching "barber" and "hairdresser") in addition to equality-matching, numeric range-matching, regex-matching, and conjunctions and disjunctions over them all.

Private Information Retrieval (PIR) and Secure Multiparty Computation (SMC). PIR allows a user to retrieve an item from a database server without revealing which item they are retrieving. Single-server PIR schemes necessarily have $\Omega(n)$ computational cost in n — the size of the database, and $\mathcal{O}(log^2 n)$ communication cost [7, 16], which is prohibitively large. The best known multi-server scheme still has a communication cost of $\mathcal{O}(n^{\frac{1}{\log \log n}})$ [43], which is still impractical in our setting. SMC is known to be harder than PIR [10]. Koi's privacy model differ from PIR and SMC, thus allowing for a much more efficient protocol that has constant communication and computational overhead.

12 Summary

We have presented Koi, a platform for supporting location-based applications in a privacy-preserving manner. Overall we find there is much to be optimistic about in raising the level of abstraction from a get lat-long API to a matching-based API. We have presented a privacy-preserving matching service, verified the privacy properties using the ProVerif theorem-prover, shown how a wide range of applications can be built using the API including implementing two concrete applications, publicly deployed the service and released client libraries, and have demonstrated the service to perform and scale well through macro- and micro-benchmarks.

References

[1] Apple iOS 4 Core Location API. http://tinyurl.com/49fyamf.

[2] Location Interoperability Forum (LIF) Privacy Guidelines. LIF TR 101, Sep 2002, http://tinyurl.com/4csy4rx.

[3] J. Bethencourt, A. Sahai, and B. Waters. Ciphertext-Policy Attribute-Based Encryption. In *IEEE Symp. on Security and Privacy*, 2007.

[4] P. Brown, J. Bovey, and X. Chen. Context-Aware Applications: From the Laboratory to the Markeyplace. *IEEE Personal Communications*, October 1997.

[5] D. L. Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *Commun. ACM*, 24(2), 1981.

[6] J. Chester, S. Grant, J. Kelsey, J. Simpson, L. Tien, M. Ngo, B. Givens, E. Hendricks, A. Fazlullah, and P. Dixon. Letter to the House Committee on Energy and Commerce. `http://tinyurl.com/y85h98g`, September 2009.

[7] B. Chor, O. Goldreich, E. Kushilevitz, and M. Sudan. Private Information Retrieval. October 1995.

[8] C. Cornelius et al. AnonySense: Privacy-Aware People-Centric Sensing. In *ACM MobiSys*, 2008.

[9] K. S. Cortier V. *Formal Models and Techniques for Analyzing Security Protocols*. IOS Press, 2010.

[10] R. Cramer, I. Damgård, and S. Dziembowski. On the complexity of verifiable secret sharing and multiparty computation. In *Proceedings of the thirty-second annual ACM symposium on Theory of computing*, 2000.

[11] D. Dolev and A. Yao. On the security of public key protocols. *Information Theory, IEEE Transactions on*, 29(2):198 – 208, March 1983.

[12] C. Dong, G. Russello, and N. Dulay. Shared and Searchable Encrypted Data for Untrusted Servers. In *DBSec*, 2008.

[13] C. Dwork. Differential Privacy. In *ICALP*, 2006.

[14] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *OSDI*, 2010.

[15] M. J. Freedman, K. Nissim, and B. Pinkas. Efficient Private Matching and Set Intersection. In *EUROCRYPT*, 2004.

[16] C. Gentry and Z. Ramzan. Computational Private Information Retrieval with Logarithmic Total Communications. July 2005.

[17] W. W. Gorlach, A. Terpstra, and A. Heinemann. Survey on Location Privacy in Pervasive Computing. In *SPPC*, 2004.

[18] V. Goyal, O. Pandey, A. Sahai, and B. Waters. Attribute-based Encryption for Fine-grained Access Control of Encrypted Data. In *ACM CCS*, 2006.

[19] M. Gruteser and D. Grunwald. Anonymous Usage of Location-Based Services through Spatial and Temporal Cloaking. In *Proceedings of MobiSys'03*.

[20] M. Gruteser and D. Grunwald. Anonymous Usage of Location-Based Services through Spatial and Temporal Cloaking. In *MobiSys*, 2008.

[21] J. Hightower, B. Brumitt, and G. Borriello. The Location Stack: A Layered Model for Location in Ubiquitous Computing. In *IEEE WMCSA*, 2002.

[22] B. Hoh, M. Gruteser, R. Herring, J. Ban, D. Work, J.-C. Herrera, A. M. Bayen, M. Annavaram, and Q. Jacobson. Virtual trip lines for distributed privacy-preserving traffic monitoring. In *MobiSys*, 2008.

[23] B. Hoh, M. Gruteser, H. Xiong, and A. Alrabady. Preserving Privacy in GPS Traces via Density-Aware Path Cloaking. In *Proceedings of CCS '07*.

[24] J. I. Hong, G. Boriello, J. A. Landay, D. W. Mcdonald, B. N. Schilit, and J. D. Tygar. Privacy and Security in the Location-enhanced World Wide Web. In *Ubicomp*, 2003.

[25] M. Ion, G. Russello, and B. Crispo. Providing Confidentiality in Content-based Publish/Subscribe Systems. In *SECRYPT*, 2010.

[26] S. Jaiswal and A. Nandi. Trust No One: A Decentralized Matching Service for Privacy in Location Based Services. In *MobiHeld*, 2010.

[27] H. Khurana. Scalable Security and Accounting Services for Content-based Publish/Subscribe Systems. In *ACM Symposium on Applied Computing*, 2005.

[28] U. Leonhardt and J. Magee. Multi-sensor location tracking. In *Mobicom*, 1998.

[29] A. Machanavajjhala, D. Kifer, J. Gehrke, and M. Venkitasubramaniam. l-diversity: Privacy beyond k-anonymity. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 1(1), March 2007.

[30] G. Myles, A. Friday, and N. Davies. Preserving Privacy in Environments with Location-Based Applications. *IEEE Pervasive Computing*, 2(1), January 2003.

[31] A. Pfitzmann and M. Köhntopp. Anonymity, unobservability, and pseudeonymity &#8212; a proposal for terminology. In *International workshop on Designing privacy enhancing technologies: design issues in anonymity and unobservability*, pages 1–9, New York, NY, USA, 2001. Springer-Verlag New York, Inc.

[32] S. Pohlig and M. Hellman. An Improved Algorithm for Computing Logarithms over gf(p) and its Cryptographic Significance. *IEEE Transactions on Information Theory*, 24(1):106–110, January 1978.

[33] Proverif. http://www.proverif.ens.fr/.

[34] C. Raiciu and D. S. Rosenblum. Enabling Confidentiality in Content-Based Publish/Subscribe Infrastructures. In *IEEE SecureComm*, 2006.

[35] M. Ryan and B. Smyth. *Formal Models and Techniques for Analyzing Security Protocols*, chapter Applied pi calculus. IOS Press, 2010.

[36] M. Scipioni and M. Langheinrich. I'm Here! Privacy Challenges in Mobile Location Sharing. In *IWSSI/SPMU*, May 2010.

[37] A. Shikfa, M. Onen, and R. Molva. Privacy-Preserving Content-Based Publish/Subscribe Networks. In *IFIP SEC*, 2009.

[38] M. Spreitzer and M. Theimer. Providing Location Information in a Ubiquitous Computing Environment. In *SOSP*, 1993.

[39] M. Srivatsa and L. Liu. Secure Event Dissemination in Publish-Subscribe Networks. In *ICDCS*, 2007.

[40] L. Sweeney. k-Anonymity: A Model for Protecting Privacy. *International Journal on Uncertainty, Fuzziness and Knowledge-based Systems*, 10(5):557–570, 2002.

[41] S. A. Weis. *New foundations for efficient authentication, commutative cryptography, and private disjointness testing*. PhD thesis, Cambridge, MA, USA, 2006. AAI0810110.

[42] C. K. Wong, M. G. Gouda, and S. S. Lam. Secure Group Communications using Key Graphs. *IEEE/ACM Transactions on Networking (TON)*, 8(1):16–30, February 2003.

[43] S. Yekhanin. New Locally Decodable Codes and Private Information Retrieval Schemes . *Electronic Colloquium on Computational Complexity*, 2006(127), 2006.

Aiding the Detection of Fake Accounts in Large Scale Social Online Services

Qiang Cao † Duke University qiangcao@cs.duke.edu
Michael Sirivianos ‡ Telefonica Research msirivi@tid.es
Xiaowei Yang Duke University xwy@cs.duke.edu
Tiago Pregueiro Tuenti, Telefonica Digital tiago@tuenti.com

Abstract

Users increasingly rely on the trustworthiness of the information exposed on Online Social Networks (OSNs). In addition, OSN providers base their business models on the marketability of this information. However, OSNs suffer from abuse in the form of the creation of fake accounts, which do not correspond to real humans. Fakes can introduce spam, manipulate online rating, or exploit knowledge extracted from the network. OSN operators currently expend significant resources to detect, manually verify, and shut down fake accounts. Tuenti, the largest OSN in Spain, dedicates 14 full-time employees in that task alone, incurring a significant monetary cost. Such a task has yet to be successfully automated because of the difficulty in reliably capturing the diverse behavior of fake and real OSN profiles.

We introduce a new tool in the hands of OSN operators, which we call *SybilRank* . It relies on social graph properties to rank users according to their perceived likelihood of being fake (Sybils). SybilRank is computationally efficient and can scale to graphs with hundreds of millions of nodes, as demonstrated by our Hadoop prototype. We deployed SybilRank in Tuenti's operation center. We found that ∼90% of the 200K accounts that SybilRank designated as most likely to be fake, actually warranted suspension. On the other hand, with Tuenti's current user-report-based approach only ∼5% of the inspected accounts are indeed fake.

1 Introduction

The surge in popularity of online social networking (OSN) services such as Facebook, Twitter, Digg, LinkedIn, Google+, and Tuenti has been accompanied by an increased interest in attacking and manipulating them. Due to their open nature, they are particularly vulnerable to the Sybil attack [26], under which a malicious user can create multiple fake OSN accounts.

The problem. It has been reported that 1.5 million fake or compromised Facebook accounts were on sale during February 2010 [7]. Fake (Sybil) OSN accounts can be used for various purposes [6, 7, 9]. For instance, they enable spammers to abuse an OSN's messaging system to post spam [28, 50], or waste an OSN advertising customer's resources by making him pay for online ad clicks or impressions from or to fake profiles. Fake accounts can also be used to acquire users' private contact lists [17]. Sybils can use the "+1" button to manipulate Google search results [11] or to pollute location crowdsourcing results [8]. Furthermore, fake accounts can be used to access personal user information [27] and perform large-scale crawls over social graphs [42].

The challenge. Due to the multitude of the reasons behind their creation (§2.1), real OSN Sybils manifest numerous and diverse profile features and activity patterns. Thus, automated Sybil detection (e.g., Machine-Learning-based) does not yield the desirable accuracy (§2.2). As a result, adversaries can cheaply create fake accounts that are able to evade detection [19]. At the same time, although the research community has extensively discussed social-graph-based Sybil defenses [23, 51–53, 56, 57], there is little evidence of wide industrial adoption due to their shortcomings in terms of effectiveness and efficiency (§2.4). Instead, OSNs employ time-consuming manual account verification, driven by user reports on abusive accounts. However, only a small fraction of the inspected accounts are indeed fake, signifying inefficient use of human labor (§2.3).

If an OSN provider can detect Sybil nodes in its system effectively, it can improve the experience of its users and their perception of the service by stemming annoying spam messages and invitations. It can also increase the marketability of its user base and its social graph. In addition, it can enable other online services or distributed systems to treat a user's online social network identity as an authentic digital identity, a vision foreseen by recent efforts such as Facebook Connect [2].

We therefore aim to answer the question: "can we design a social-graph-based mechanism that enables a multi-million-user OSN to pick up accounts that are very likely to be Sybils?" The answer can help an OSN stem malicious activities in its network. For example, it can enable human verifiers to focus on likely-to-be-fake accounts. It can also guide the OSN in sending CAPTCHA [14] challenges to suspicious accounts, while running a lower risk to annoy legitimate users.

Our solution. We present the design (§4), implementation (§5), and evaluation (§6,§7,§5) of SybilRank. SybilRank is a Sybil inference scheme customized for OSNs whose social relationships are bidirectional.

†Part of Q. Cao's work was conducted while interning with Telefonica Research.
‡M. Sirivianos is currently with the Cyprus University of Technology.

Our design is based on the same assumption as in previous work on social-graph-based defenses [23, 51–53, 56, 57]: that OSN Sybils have a disproportionately small number of connections to non-Sybil users. Differently, our system achieves a significantly higher detection accuracy at a substantially lower computational cost. It can also be implemented in a parallel computing framework, e.g., MapReduce [24], enabling the inference of Sybils in OSNs with hundreds of millions of users.

Social-graph-based solutions uncover Sybils from the perspective of already known non-Sybil nodes (*trust seeds*). Unlike [51] and [52], SybilRank's computational cost does not increase with the number of trust seeds it uses (§4.5). This facilitates the use of multiple seeds to increase the system's robustness to seed selection errors, such as designating a Sybil as a seed. It also allows the OSN to cope with the multi-community structure of online social graphs [53] (§4.2.2).

We show that SybilRank outperforms existing approaches [23,33,38,52,56]. In our experiments, it detects Sybils with at least 20% lower false positive and negative rates (§6) than the second-best contender in most of the attack scenarios. We also deployed SybilRank on Tuenti, the leading OSN in Spain (§7). Almost 100% and 90% of the 50K and 200K accounts, respectively, that SybilRank designated as most suspicious, were indeed fake. This compares very favorably to the ∼5% hit rate of Tuenti's current abuse-report-based approach.

Our contributions. In summary, this work makes the following contributions:

• We re-formulate the problem of Sybil detection in OSNs. We observe that due to the high false positives of binary Sybil/non-Sybil classifiers, manual inspection needs to be part of the decision process for suspending an account. Consequently, our aim is to efficiently derive a quality ranking, in which a substantial portion of Sybils ranks low. This enables the OSN to focus its manual inspection efforts towards the end of the list, where it is more likely to encounter Sybils. Moreover, our ranking can inform the OSN on whether to challenge suspicious users with CAPTCHAs.

• The design of SybilRank, an effective Sybil-likelihood ranking scheme for OSN users based on the landing probability of short random walks. We efficiently compute this landing probability using early terminated power iteration. In addition, SybilRank copes with the multi-community structure of social graphs [53] using multiple seeds at no additional computational cost. We thoroughly compare SybilRank with existing Sybil detection schemes.

• We deployed SybilRank on Tuenti, an OSN with 11M users. Our system has allowed Tuenti operations to detect in the same time period 18 times more fake accounts than their current process.

2 Background and Related Work

We have conducted a survey with Tuenti [4] to understand the activities of fake accounts in the real world, the importance of the problem for them, and what countermeasures they currently employ. In the rest, we discuss our findings and prior OSN Sybil defense proposals.

2.1 Fake accounts in the real world

Fake accounts are created for profitable malicious activities, such as spamming, click-fraud, malware distribution, and identity fraud [7,12]. Some fakes are created to increase the visibility of niche content, forum posts, and fan pages by manipulating votes or view counts [6,9].

People also create fake profiles for social reasons. These include friendly pranks, stalking, cyberbullying, and concealing a real identity to bypass real-life constraints [9]. Many fake accounts are also created for social online games [9].

2.2 Feature-based approaches

The large number of distinct reasons for the creation of Sybil accounts [6, 7, 9] results in numerous and diverse malicious behaviors manifested as profile features and activity patterns. Automated feature-based Sybil detection [54] (e.g., Machine-Learning-based) suffers from high false negative and positive rates due to the large variety and unpredictability of legitimate and malicious OSN users' behaviors. This is reminiscent of how ML has not been very effective in intrusion detection [49].

High false positives in particular pose a major obstacle in the deployment of automated methods, as real users respond very negatively to erroneous suspension of their accounts [10]. To make matters worse, attackers can always adapt their behavior to evade detection by automated classifiers. Boshmaf et al. [19] recently showed that the automated Facebook Immune System was able to detect only 20% of the fakes they deployed, and almost all the detected accounts were flagged by users.

2.3 Human-in-the-loop counter-measures

In response to the inapplicability of automated account suspension, OSNs employ CAPTCHA [5] and photo-based social authentication [13] to rate-limit suspected users, or manually inspect the features of accounts reported as abusive (flagged) by other users [12,50,54].

The manual inspection involves matching profile photos to the age or address, understanding natural language in posts, examining the friends of a user, etc [12]. The inspectors also use simple tools to parse activity and IP statistics of suspicious accounts. However, these tasks require human intelligence and intuition, rendering them hard to automate and scale. In addition, flagged accounts have by definition already caused annoyance to some users. Therefore, the need arises for a method that en-

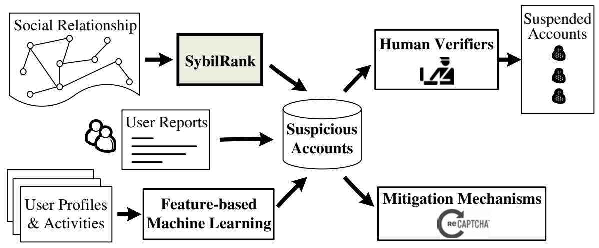


Figure 1: **SybilRank as a part of an OSN's Sybil defense toolchain.**

ables human verifiers to focus on accounts that are very likely to be fake and to disable them in a timely manner.

Tuenti receives on average 12,000 reports regarding abusive accounts and 4,000 reports for violating photos per day. An employee can review an average of 250 to 350 reports in an hour. Among those reported suspicious accounts, only $\sim$5% are indeed fake [12]. On average, Tuenti's manual inspection team, which consists of 14 employees, deletes 800 to 1500 fake accounts per day.

Our solution can be a component of the overall framework employed by OSN providers to ensure the health of their user base (Figure 1). It is a proactive method that can uncover fakes even before they interact with real users. It complements ML- and user-report-based methods. The users that these methods designate as suspicious are typically challenged in the form of CAPTCHAs or are manually inspected.

2.4 Social-graph-based approaches

Sybil detection has been the focus of the research community for a while now [23, 51–53, 56, 57]. However, *none of the existing proposals can play the role of SybilRank in Figure 1*. The reason is that prior schemes either do not exhibit equivalent accuracy or they incur a prohibitive computational cost. Their key features are documented in a recent survey [55]. We next compare them to our proposal.

The decentralized protocols SybilGuard [57] and SybilLimit [56] infer Sybils based on a large volume of random walk traces, which leads to an $O(\sqrt{mn}\log n)$ (m is the number of edges, n is the number of nodes) computational cost in a centralized setting. SybilInfer [23] is also built on random walk traces with $O(n(\log n)^2)$ cost per honest seed, but does not specify any upper-bound guarantee on false rates [55]. Mohaisen et al. [40] propose to use weighted random walks in SybilLimit. However, they follow the random walk sampling paradigm instead of our power-iteration method (§4.2), incurring a cost as high as SybilLimit.

The flow-based scheme GateKeeper [52] improves over SumUp [51]. It relies on a strong assumption that requires balanced graphs [55] and costs $O(sn\log n)$ (s is the number of seeds, referred to as ticket sources). Viswanath et al. [53] propose community detection algorithms such as Mislove's algorithm [38] to detect Sybils. However, community detection (CD) algorithms rarely provide provable guarantees, and Mislove's CD algorithm costs $O(n^2)$. Compared to the above schemes, SybilRank achieves equivalent or higher accuracy (analytically), but it is computationally more efficient, i.e., it has $O(n\log n)$ cost irrespective of the number of seeds.

Moreover, SybilRank addresses important limitations of existing social-graph-based defenses [53]. First, SybilRank leverages its efficient support for multiple trust seeds to reduce the false positives resulting from the existence of multiple non-Sybil communities. Second, it enables very flexible seed selection (any non-Sybil node can be a seed), which makes it harder for attackers to target the seeds. In addition, SybilRank's effectiveness decreases only moderately as the Sybil's distance from the trust seeds decreases (§6.5).

Zhao et al.'s [60] BotGraph detects spam webmail user-bots by leveraging the fact that they are highly correlated in terms of the IP address of their controllers. The same mechanism can be used in social graphs to uncover Sybils that form tight-knit communities. Tangentially, Yang et al. [54] recently analyzed a sample of 660K Sybils in the RenRen OSN, and found that they do not form tight-knit communities. Regardless, SybilRank and the schemes in [23, 52, 53, 57] rely on the assumption that the connectivity between Sybils and non-Sybil users is limited and are not very sensitive to the number of Sybils or the inter-Sybil connectivity.

Sybil-resilient systems [27, 36, 37, 42, 44, 46–48, 58] leverage application-specific knowledge to effectively mitigate Sybils, e.g., to limit the number of votes collected from Sybil users [51]. However, a Sybil-resilient design optimized for an application may not be applicable to other systems, while a social-graph-based Sybil inference mechanism can be applied to any application that binds its users to OSN accounts.

3 Models, Assumptions, and Goals

We now introduce our system and threat model, and our design assumptions and goals.

3.1 System and threat model

We consider bilateral social relationships and model an OSN as an *undirected* graph $G = (V, E)$, where each node in V corresponds to a user in the network, and each edge in E corresponds to a bilateral social relationship. In the social graph G, there are $n = |V|$ nodes, $m = |E|$ undirected edges, and a node v has a degree of $deg(v)$. We assume that the OSN provider, e.g., Tuenti, has access to the entire social graph.

In our threat model, attackers can launch Sybil attacks [21, 26] by creating many fake identities. Like ex-

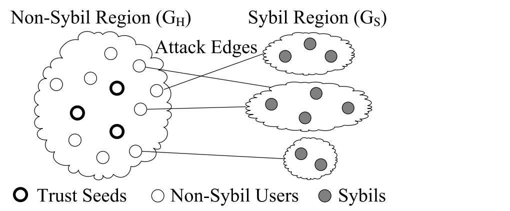


Figure 2: Non-Sybil region, Sybil region, and attack edges in an OSN under a Sybil attack. All Sybils created by malicious users are placed into the Sybil region. The Sybil collective may not be well connected.

isting work [23,56,57], we divide the node set V into two disjoint sets H and S, representing non-Sybil and Sybil users respectively, as shown in Figure 2. We denote the *non-Sybil region* G_H as the subgraph induced by the non-Sybil user set H, which includes all non-Sybil users and the links among them. Similarly, the *Sybil region* G_S is the subgraph induced by S. The non-Sybil and the Sybil regions are connected by g *attack edges* between Sybils and non-Sybil users.

3.2 Assumptions

We make the following assumptions.

Social graph. The social graph is undirected. The non-Sybil region G_H is well connected and non-bipartite. In this case, random walks on the graph can be modeled as an irreducible and aperiodic Markov chain [16]. This Markov chain is guaranteed to converge to a stationary distribution in which the landing probability on each node after sufficient steps is proportional to its degree.

Limited attack edges. We assume that Sybils establish a limited number of attack edges due to the difficulty of soliciting and maintaining reciprocal social relationships. Although previous studies [17, 19] suggest that fake identities can befriend others, a recent study shows that most of their connections are established with other fakes [43]. SybilRank is designed for large scale attacks, where fake accounts are crafted and maintained at a low cost, and are thus unable to befriend many real users. Furthermore, SybilRank can be deployed over a social graph that includes only strong-relationship edges. For instance, Google+ may consider only social connections between users that appear in mutually close circles.

The limited attack edges result in a sparse cut between the non-Sybil and the Sybil region. Since the well-connected non-Sybil region is unlikely to have such a sparse cut [57], there should be a significant difference between the mixing time of the non-Sybil region G_H and the entire graph G [16]. A graph's *mixing time* is the maximum number of steps that a random walk needs to make so that the probability of landing at each node reaches the stationary distribution [16]. As in previous work [23, 56, 57], we start with the assumption that the non-Sybil region G_H is fast mixing, i.e., its mixing time is $O(\log n)$, and discuss the scenarios where the non-Sybil region mixes slower in §4.2.3.

Attack edge distribution. To make our analysis tractable, we assume that Sybils randomly attach attack edges to non-Sybils. A more effective attack strategy against a social-graph-based Sybil defense is to establish attack edges close to the trust seeds. We refer to this attack as a *targeted attack*. We experimentally study how SybilRank performs under the targeted attack in § 6.5.

3.3 Goals

SybilRank aims to aid OSNs in identifying highly suspicious accounts by ranking users. In principle, SybilRank can be used to defend against malicious accounts that are created on a large scale and at a low per-account cost for purposes, such as rating manipulation (e.g., with Facebook "likes"), spam, and crawls. Our design has the following main goals:

Effectiveness. The system should mostly rank nodes that are Sybils lower than non-Sybils (low false positives), while limiting the number of non-Sybils ranked below Sybils (low false negatives). It should be robust under various attack strategies. A very high portion of the nodes at the bottom of the ranked list should be fake. The portion of fakes can decrease as we go up the list.

Efficiency. The system should have a low computation cost. It should be able to handle large social networks with commodity machines, so that OSN providers can deploy it on their existing clusters.

4 Design

We now describe the design of SybilRank. We first provide an overview of the approach and proceed with a detailed description of each component.

4.1 Overview

SybilRank relies on the observation that an *early-terminated* random walk [55] starting from a non-Sybil node in a social network has a higher degree-normalized (divided by the degree) landing probability to land at a non-Sybil node than a Sybil node. Intuitively, this observation holds because the limited number of attack edges forms a narrow passage from the non-Sybil region to the Sybil region in a social network. When a random walk is sufficiently long, it has a uniform degree-normalized probability of landing at any node, a property referred to as the convergence of a random walk [16]. However, a shortened random walk originating from a non-Sybil node tends to stay within the non-Sybil region of the network, as the walk is unlikely to traverse one of the relatively few attack edges.

Our key insight is to rank nodes in a social graph according to the degree-normalized probability of a short random walk that starts from a non-Sybil node to land on them. We screen out low-ranked nodes as potential fake users. A further novelty of our approach is that we use power iteration [34], a standard technique to efficiently calculate the landing probability of random walks in large graphs. This is in contrast to prior work that used a large number of random walk traces [23,40,55–57] obtained at a high computational cost. For ease of exposition, we refer to the probability of the random walk to land on a node as the node's *trust*.

As shown in Figure 3, SybilRank unveils users that are suspected to be Sybils after three stages. In Stage I, through $w = O(\log n)$ power iterations (§4.2), trust flows from known non-Sybil nodes (trust seeds) and spreads over the entire network with a bias towards the non-Sybil region. In Stage II, SybilRank ranks nodes based on their degree-normalized trust. In the final stage, SybilRank assigns portions of fake nodes in the intervals of the ranked list. This enables OSNs to focus their manual inspection efforts or to regulate the frequency with which they send CAPTCHAs to suspected users.

We next describe each stage in detail.

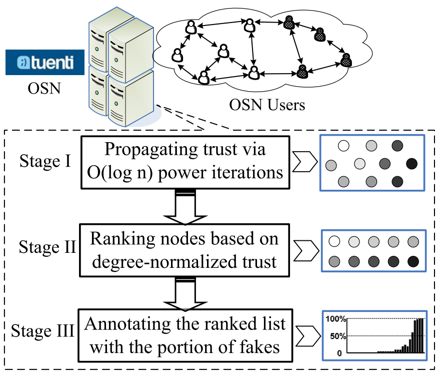

tuenti
OSN
OSN Users
Stage I
Propagating trust via O(log n) power iterations
Stage II
Ranking nodes based on degree-normalized trust
Stage III
Annotating the ranked list with the portion of fakes
100%
50%
0

Figure 3: SybilRank detects Sybils in three stages. Black users are Sybils. Darker nodes obtain less trust after trust propagation.

4.2 Propagating trust

Power iteration involves successive matrix multiplications where each element of the matrix represents the random walk transition probability from one node to a neighbor node. Each iteration computes the landing probability distribution over all nodes as the random walk proceeds by one step.

4.2.1 Early-terminated random walks

We terminate the power iterations after $O(\log n)$ steps. The number of iterations needed to reach the stationary distribution is equal to the graph's mixing time. In an undirected graph, if a random walk's transition probability to a neighbor node is uniformly distributed, the landing probability on each node remains proportional to its degree after reaching the stationary distribution. SybilRank exploits the mixing time difference between the non-Sybil region G_H and the entire graph G to distinguish Sybils from non-Sybils. The intuition is that if we seed all trust in the non-Sybil region, then trust can flow into the Sybil region only via the limited number of attack edges. If we terminate the power iteration early before it converges globally, non-Sybil users will obtain higher trust than that in the stationary distribution, whereas the reverse holds for Sybils.

Trust propagation via power iteration. We define $T^{(i)}(v)$ as the trust value on node v after i iterations. Initially, the total trust, denoted as T_G ($T_G > 0$), is evenly distributed on K ($K > 0$) trust seeds $\tilde{v}_1, \tilde{v}_2, \ldots, \tilde{v}_K$:

$$T^{(0)}(v) = \begin{cases} \frac{T_G}{K} & \text{if node } v \text{ is one of the } K \text{ trust seeds} \\ 0 & \text{else} \end{cases}$$

Seeding trust on multiple nodes makes SybilRank robust to seed selection errors, as incorrectly designating a node that is Sybil or close to Sybils as a seed causes only a small fraction of the total trust to be initialized in the Sybil region.

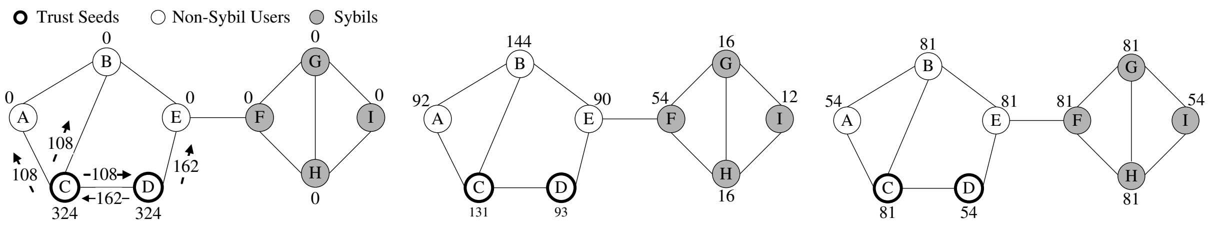
During each power iteration, a node first evenly distributes its trust to its neighbors. It then collects trust distributed by its neighbors and updates its own trust accordingly. The process is shown below. Note that the total amount of trust T_G remains unchanged.

$$T^{(i)}(v) = \sum_{(u,v)\in E} \frac{T^{(i-1)}(u)}{deg(u)}$$

Early termination. With sufficiently many power iterations, the trust vector converges to the stationary distribution: $\lim_{i\to\infty} T^{(i)}(v) = \frac{deg(v)}{2m} \times T_G$ [16]. However, SybilRank terminates the power iteration after $w = O(\log n)$ steps, thus before convergence. This number of iterations is sufficient to reach an approximately uniform distribution of degree-normalized trust over the fast-mixing non-Sybil region, but limits the trust escaping to the Sybil region.

Alternative use of power iteration. We note that power iteration is also used by the trust inference mechanisms PageRank, EigenTrust, and TrustRank [22, 30, 33, 45], where it is executed until convergence to the stationary distribution of the complete graph [34]. At each step and with a constant probability, PageRank's random walks jump to random users, and EigenTrust/TrustRank's walks jump to trust seeds. EigenTrust/TrustRank is personalized PageRank with a customized reset distribution over a particular set of seeds.

SybilRank's random walks do not jump to random users, because this would allow Sybils to receive trust in each iteration. Besides, SybilRank's random walks do not jump to the trust seeds, because this would assign high trust to Sybils that are "close" to the trust seeds (§6.3).



(a) Initialization (b) After 4 power iterations (c) Stationary distribution

Figure 4: **Trust propagation through power iterations.**

In addition, we do not seek to improve the Sybil resilience of power-iteration-based trust inference mechanisms by enforcing early termination. This is because with their implicit directed connections to random nodes (PageRank) and trust seeds (EigenTrust and TrustRank), the mixing time of their modified graphs decreases significantly. A formal analysis on the convergence of random walks on these graphs is documented in [34]. Empirical reports show that EigenTrust only takes 10 iterations to converge on a 1000-node graph [33]. Therefore, early termination after $O(\log n)$ steps cannot improve those schemes.

Example. Figure 4 illustrates the process of our trust propagation. In this example, we initially seed $T_G = 648$ on the non-Sybil nodes C and D. We do not normalize T_G to 1 for ease of exposition. After four power iterations, all non-Sybil nodes $\{A, B, C, D, E\}$ obtain higher degree-normalized trust than any of the Sybils $\{F, G, H, I\}$. However, as shown in Figure 4(c), if we let the power iteration converge (more than 50 iterations), the Sybils have similar trust as the non-Sybil nodes, and each node's trust is only proportional to its degree.

4.2.2 Coping with the multi-community structure

Existing social-graph-based defenses are sensitive to the multi-community structure in the non-Sybil region [53]. Due to the limited connectivity between communities, they may misclassify non-Sybils that do not belong to the communities of the trust seeds as Sybils.

Distributing seeds among communities. SybilRank intends to reach a uniform distribution of degree-normalized trust among the non-Sybil region, which is independent of the location of the non-Sybil seeds. Thus, any set of non-Sybil users are eligible for the trust seeds. Seeding trust on Sybils would degrade SybilRank's effectiveness as the initial trust on the Sybil seeds is not bounded by the limited attack edges. OSN providers can easily identify non-Sybil users to seed trust by manually inspecting a few users. SybilRank works with an arbitrary number of non-Sybil seeds regardless of their locations. In contrast, the seed selection is complicated for previous trust inference schemes (§4.2.1) and Sybil defenses such as SumUp [51] because they have to guarantee that the seed(s) are "far" from Sybils.

We leverage SybilRank's support for multiple seeds to improve its performance in OSNs that have a multi-community structure. The key idea is to distribute seeds among the communities. Thus, at initialization we distribute trust in such a manner that the non-Sybil users are not starved of trust even if the inter-community connectivity is weak. We validate this design choice with simulations in §6.4, where SybilRank maintains a high detection accuracy in synthetic graphs with high-level community structure.

Inspecting random users for trust seeds to cover most of communities in the non-Sybil region would require a prohibitive manual inspection workload, as indicated by the Coupon Collector's Problem [39]. Instead, we apply an efficient community detection but not Sybil-resilient algorithm to obtain an estimation of the community structure [18], and then seed trust on non-Sybil users in each major community.

Estimating the multi-community structure. We use the Louvain method [18], which can efficiently detect communities. This method iteratively groups closely connected communities together to improve the partition modularity. In each iteration, every node represents a community, and well-connected neighbor nodes are combined into the same community. The graph is reconstructed at the end of each iteration by converting the resulting communities to nodes and adding links that are weighted by the inter-community connectivity. Each iteration has a computational cost linear to the number of edges in the corresponding graph and a small number of iterations are typically required. In addition, this method can be parallelized on commodity machines [59].

As illustrated in Figure 5, in each identified community we inspect a small set of random nodes ($\sim$100 in total for the 11-million-node Tuenti social network) and only seed trust at the nodes that pass the verification.

Community detection algorithms have been proposed to directly detect Sybils [53]. They seek a partition of a graph that has dense intra-community connectivity and weak inter-community connectivity. For instance, the Louvain method searches for a partition with high modularity [18]. Thus, Sybils that are not well connected among each other may be classified as non-Sybils belonging to nearby non-Sybil communities. In contrast,

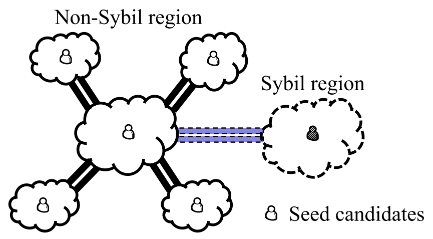


Figure 5: Distributing seeds into multiple communities. The Sybil seed candidates (black) cannot pass the manual inspection.

by placing seeds in communities SybilRank is able to uncover subsets of Sybils within Louvain-detected communities, as shown in our Tuenti deployment (§7.2).

4.2.3 Sensitivity to the mixing time

Although for analytical tractability, we assume that the non-Sybil region is fast mixing as in previous work [23, 52, 56], SybilRank's effectiveness does not rely on the absolute value of the non-Sybil region's mixing time. Instead, it only requires that the graph G containing both Sybils and non-Sybils has a longer mixing time than the non-Sybil region G_H. Under this condition, the early-terminated power iteration yields a gap between the degree-normalized trust of non-Sybils and Sybils.

Ideally, the number of iterations that SybilRank performs is set equal to the mixing time of the non-Sybil region. Previous Sybil defenses [23,56] assume that the mixing time of social networks is $O(\log n)$. However, measurement studies [25,41] show that the mixing time of some social networks is longer than expected. It is unclear whether this increase derives from a large coefficient in front of the term $\log n$, or the possibility that the mixing time of social networks is not $O(\log n)$.

In SybilRank we simply use $O(\log n)$ power iterations. If the mixing time of the non-Sybil region is larger than this value, the trust that escapes to the Sybil region is further limited. However, we also run the risk of starving the non-Sybil users that are not well-connected to seeds. This risk is mitigated by placing seeds in many communities and by dispersing multiple seeds in each community (§4.2.2), thereby ensuring that the trust is initiated somewhere close to those non-Sybil users.

4.3 Ranking by degree-normalized trust

After $w = O(\log n)$ power iterations, SybilRank ranks nodes by their degree-normalized trust. A node v's degree-normalized trust is defined as: $\hat{T}_v = \frac{T^{(w)}(v)}{deg(v)}$. We rank nodes by degree-normalized trust for the following two reasons.

Eliminating the node degree bias. This design gives each non-Sybil node almost identical degree-normalized trust. It reduces false positives from low-degree non-Sybil nodes and false negatives from high-degree Sybils. This is because after $O(\log n)$ power iterations, the trust distribution in the non-Sybil region approximates the stationary distribution in that region, i.e., the amount of trust on each non-Sybil node is proportional to its degree.

The removal of the degree bias simplifies the Sybil/non-Sybil classification, i.e., all non-Sybil users are supposed to belong to the same class with an almost identical degree-normalized trust. This is in contrast to prior trust inference schemes [30,33,45] that attempt to differentiate between Sybils and non-Sybils with highly variable trust scores.

Bounding highly-ranked Sybils. The degree normalization step ensures that the number of Sybils ranked higher than the non-Sybil nodes is $O(\log n)$ per attack edge, as we explain in §4.6.

4.4 Annotating the ranked list

SybilRank relies on the limited attack edges to distinguish Sybils. It may give high rankings to the Sybils that obtain many social links to non-Sybil users, and consider them as non-Sybil users. This is true for all social-graph-based defenses. It may result in even the top and bottom intervals of the ranked list to comprise a non-negligible portion of fake and real accounts, respectively. Thus, an OSN cannot simply identify a pivot in the ranked list below which all nodes are fake, and it still has to rely on manual inspection.

At the same time, OSNs have limited resources for manual inspection. To aid OSNs to adjust their inspection focus, we annotate intervals in the ranked list with fake portions. In particular, we sample random users in each interval of a particular size and report a portion of fakes after manual inspection. With these annotations, OSNs can decide where on the ranked list to assign their limited human verifiers. The annotations can also be used to regulate the frequency of CAPTCHAs and other challenges sent to the suspected users. Moreover, the annotations can help OSNs to decide on accounts that do not exhibit sufficient evidence on whether they are fake.

4.5 Computational cost

SybilRank's computational cost is $O(n \log n)$. This is because each power iteration costs $O(n)$, and we iterate $O(\log n)$ times. The cost for ranking the nodes according to their degree-normalized trust is also $O(n \log n)$. The cost for estimating communities is $O(m)$, because each iteration in Louvain method has a computational cost linear to the number of edges and the graph shrinks rapidly with only a few iterations. Since the node degree in OSNs is always limited, the community estimation costs $O(n)$. Thus the overall computational cost is $O(n \log n)$, irrespective of the number of trust seeds.

4.6 Security Guarantee

We now discuss SybilRank's security guarantee under the assumption that the attack edges are randomly established between non-Sybils and Sybils. Although our system does not depend on the absolute mixing time of the non-Sybil region (§4.2.3), our analysis assumes that the non-Sybil region is fast-mixing. Due to space limitations we only present the conclusion and its high-level intuition, and refer the reader to our technical report [20] for the complete proof. We use the notation in §3.1.

Theorem 1. *When an attacker randomly establishes g attack edges in a fast mixing social network, the total number of Sybils that rank higher than non-Sybils is $O(g \log n)$.*

After early termination, the trust distribution in the entire graph G has not reached its stationary distribution. Since trust propagation starts from the non-Sybil region, the Sybil region (on the aggregate) gets only a fraction $f < 1$ of the trust it should obtain in the stationary distribution. On the other hand, as the total trust is conserved in the entire graph, the non-Sybil region obtains an aggregate amount of trust that is c ($c > 1$) times higher than in the stationary distribution. Further, since the non-Sybil region is well connected, each non-Sybil node obtains approximately identical degree-normalized trust, i.e., $c \times \frac{T_G}{2m}$, where $\frac{T_G}{2m}$ is a node's degree-normalized trust in the stationary distribution (§4.2).

The amount of degree-normalized trust obtained by each Sybil depends on how Sybils connect to each other. However, since the aggregate amount of trust of the Sybil region is bounded, on average, each Sybil obtains $f \times \frac{T_G}{2m}$ degree-normalized trust, which is less than that of a non-Sybil node. We are able to show that at most $O(\log n)$ Sybils per attack edge obtain higher degree-normalized trust than non-Sybil nodes [20]. It is worth noting that SybilRank is able to provide this guarantee even when it uses an arbitrary number of trust seeds.

5 MapReduce Implementation

We now briefly describe how we implement SybilRank using the Hadoop [1] MapReduce [24] parallel computing framework. This implementation enables an OSN provider to process social network graphs with hundreds of millions of users on a cluster of commodity machines.

We divide the entire graph into multiple partitions so that each of them fits into the hardware of a commodity machine. The complexity of SybilRank is dominated by the first two stages (Figure 3): trust propagation and node ranking. Together they have $O(n \log n)$ complexity. For the trust propagation stage, we observe that trust splitting and trust aggregation at each iteration are inherently parallelizable. Therefore, we treat each iteration as a MapReduce job, and create multiple map tasks to split trust and multiple reduce tasks to aggregate trust simultaneously. For the node ranking stage, we use Hadoop's built-in sorting feature.

Efficiency. We evaluate the efficiency of our prototype on an Amazon EC2 cluster to process very large-scale synthetic graphs with hundreds of millions of nodes. The cluster consists of 11 m1.large instances, one of which serves as the master and the other 10 as slaves.

We generate very large synthetic graphs based on the scale-free model as in §6.1. The synthetic graphs are generated with exponentially increasing sizes from 10M to 160M nodes. SybilRank performs successfully on each graph with $\log n$ power iterations. The total execution time includes two parts: a) the time to seed trust and partition the graph during initialization; and b) the time to execute power iterations to propagate trust and to rank the final results. The latter dominates the total execution time, which increases almost linearly with the size of the social graphs (see [20]). For the largest graph (160M nodes), our prototype finishes in less than 33 hours. This result suggests that SybilRank can process very large social graphs using a few commodity machines.

6 Sybil Ranking Evaluation

In this section, we perform a comparative evaluation of SybilRank's ability to provide a meaningful ranking of nodes to uncover Sybils. We first compare SybilRank against other approaches in terms of its effectiveness in assigning low ranking to Sybils. We then examine the SybilRank's component that copes with the multi-community structure, and we study the resilience of our approach to attacks that target the seeds. For a fair comparison, we do not use SybilRank's component for coping with the multi-community structure except for §6.4.

6.1 Simulation setup

Compared approaches. We compare SybilRank (SR) against the state-of-the-art social-graph-based Sybil defenses, i.e., SybilLimit (SL) [56], SybilInfer (SI) [23], Mislove's community detection [38] (CD), and GateKeeper (GK) [52]. Importantly, we also compare to EigenTrust (ET) [33], which uses power iteration to assign trust.

Datasets. The non-Sybil regions of the simulated social graphs, which comprise exclusively non-Sybils, are samples of several popular social networks (Table 1). The Facebook graph [29] is a connected component sampled via the "forest fire" sampling method [35]. The synthetic graph is generated using Barabasi's scale-free model [15]. The rest of the graphs [3] have been widely used to study social graph properties and to evaluate recent Sybil defense mechanisms [41,53].

Attack strategies. We create a $5K$-node Sybil region that connects to a non-Sybil region through a varying number of random attack edges. We choose this large number of Sybils to stress-test each scheme. To investigate the schemes' robustness to the formation of the Sybil collective, we include two representative Sybil region structures: regular random graphs and scale-free graphs [15]. We call the first attack *regular attack*: each Sybil establishes connections to d random Sybils. We refer to the second attack as a *scale-free attack*: each Sybil preferentially connects to d Sybils upon its arrival, with the probability of connecting to a Sybil proportional to the Sybil's degree. We set $d = 4$ in both attacks.

Social Network	Nodes	Edges	Clustering Coefficient	Diameter
Facebook	$10,000$	$40,013$	0.2332	17
ca-AstroPh	$18,772$	$198,080$	0.3158	14
ca-HepTh	$9,877$	$25,985$	0.2734	18
Synthetic	$10,000$	$39,399$	0.0018	7
wiki-Vote	$7,115$	$100,736$	0.1250	7
soc-Epinions	$10,000$	$222,077$	0.0946	6
soc-Slashdot	$10,000$	$153,404$	0.0582	4
email-Enron	$10,000$	$105,343$	0.1159	6

Table 1: Social graphs used in our experiments. The last three graphs are 10K-node BFS samples.

Performance metrics. Our evaluation is based on a framework [53] that reduces defense schemes to a general model: producing a trust-based node ranking. The conversion for SybilLimit, SybilInfer, and CD is documented in [53]. For GateKeeper, we rank nodes by the number of tickets that each node obtains. For EigenTrust, we rank nodes according to their trust scores.

We use three metrics to compare the node ranking: the area under the Receiver Operating Characteristic (ROC) curve [31], the false positive rates, and the false negative rates. The ROC curve exhibits the change of the true positive rate with the false positive rate as a pivot point moves along the ranked list: a node below the pivot point in the ranked list is determined to be a Sybil; if the node is actually a non-Sybil, we have a false positive. The area under the ROC curve measures the overall quality of the ranking, i.e., the probability that a random non-Sybil node is ranked higher than a random Sybil. It ranges from 0 to 1, with 0.5 indicating a random ranking. An effective Sybil detection scheme should achieve a value > 0.5. Given a node ranked list, sliding the pivot point regulates the trade-off between the two false rates. We set the pivot point based on a fixed value for one false rate and compute the other false rate. We set the fixed false rate equal to 20%. In the real world, OSNs do not need a pivot point because none of the defenses so far can yield a binary Sybil/non-Sybil classifier with an acceptable false positive rate.

Trust seed selection. For a fair comparison, we strive to use the same trust seeds for all schemes in each simulation on each social network. For schemes that use a single seed, we randomly pick a node from the top-10 non-Sybil nodes that have the highest degree. For schemes supporting multiple seeds at one run, i.e., SybilRank, EigenTrust, and GateKeeper, we use 50 trust seeds. One seed is the same top-10 degree node as the one used in the single-seed schemes, and the other 49 seeds are randomly chosen from the non-Sybil nodes.

Other simulation settings. We perform $\log n$ power iterations for SybilRank, where n is the size of the social graph. We run EigenTrust until convergence with a reset probability 0.15, as in [33]. For each attack scenario, we average the results over 100 runs.

6.2 Ranking Quality Comparison

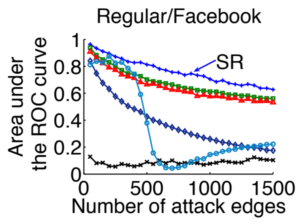
To compare the Sybil defense schemes, we start with a few attack edges and increase the number to a sufficiently large value such that the detection accuracy of each scheme degrades significantly. We show representative simulation results in Figure 6 and refer the reader to our technical report for the complete results [20]. As can be seen, when the number of attack edges is small, most of the schemes perform well and Sybils can be distinguished from non-Sybils by connectivity.

SybilRank. SybilRank outperforms all other schemes. It achieves the highest value of the area under the ROC curve and the lowest false positive and false negative rates. For the Facebook graph under the regular attack, even if the $5K$-node Sybil cluster obtains 1500 attack edges, a non-Sybil node has a probability of 70% to rank higher than a random Sybil as indicated by the value of the area under the ROC curve.

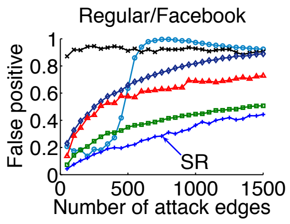
SybilLimit. SybilLimit outperforms most other schemes, but it performs worse than SybilRank. We believe that this is due to the different use of random walks, i.e., SybilLimit uses random walk traces, while SybilRank uses the power-iteration-computed landing probability. SybilLimit's security guarantee only limits the Sybils accepted by the verifier's (trust seed's) random walks that never cross any attack edge to the Sybil region [55]. However, the accepted Sybils cannot be bounded if a verifier's random walk enters into the Sybil region. In contrast, SybilRank allows trust to escape to the Sybil region, but does not accept a Sybil unless it gets higher degree-normalized trust than non-Sybil users.

GateKeeper. It performs worse than both SybilRank and SybilLimit, although it bounds the accepted Sybils to $O(\log g)$ per attack edge, where g is the total number of attack edges. This is because this bound comes from a strong assumption that does not always hold in real social networks: with high probability, a breadth-first search starting from a non-Sybil user and visiting at most $n/2$ nodes covers a large fraction of non-Sybil users [55].

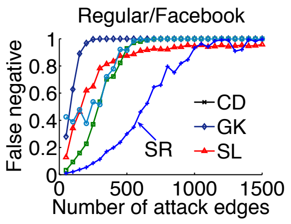
SybilInfer. We observe a steep fall in the area under the ROC curve for SybilInfer when the number of at-



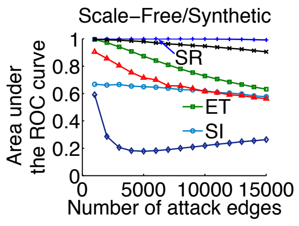
(a)



(b)



(c)



(d)

Figure 6: We depict the area under the ROC curve, the false positive rate, and the false negative rate with respect to the number of attack edges under the regular attack and the scale-free attack for various Sybil detection schemes in the Facebook and the scale-free synthetic graphs. In the ROC curve figures (a and d), a higher curve indicates a more effective scheme. In the false rate figures (b and c), a lower curve indicates a more effective scheme. SR stands for SybilRank; CD for the community detection algorithm, GK for GateKeeper; SI for SybilInfer; ET for EigenTrust; and SL for SybilLimit.

tack edges is close to 500 in Facebook under the regular attack. We suspect that this sharp performance degradation is due to the fact that SybilInfer uses the Metropolis-Hastings (MH) algorithm to sample the non-Sybil node set [23]. However, it remains unclear when the sampling converges, although Danezis et al. provide an empirical estimation and terminate the sampling after $O(n \log n)$ steps. If the Sybils obtain many attack edges and become hard to be detected, $O(n \log n)$ steps may not suffice to reach the convergence of the MH sampling and therefore, the detection accuracy is likely to degrade.

Mislove's CD. It underperforms under the regular attack, with <0.2 under the ROC curve area (Figure 6(a)). It interestingly becomes effective under the scale-free attack in the synthetic graph (Figure 6(d)). This significant performance difference is due to the greedy search for the local community, which is sensitive to the graph topology and cannot provide a false rate bound [55].

Although Mislove's CD algorithm was intended to be used with one seed, it can support multiple seeds at the same cost: by initializing the local community search with those seeds. In §6.4 we show that this extension can improve its performance to some extent.

EigenTrust. EigenTrust improves over PageRank [45], and uses the same basic mechanism as TrustRank [30]. We can see that in Figure 6(a) EigenTrust mostly outperforms previously proposed Sybil defenses. However, it has at least 20% higher false positive and negative rates than SybilRank in most of the attack scenarios.

6.3 Comparison with EigenTrust

EigenTrust is more related to SybilRank because it also uses power iteration. By further investigating EigenTrust we reveal the importance of SybilRank's two main differentiating characteristics: a) removal of degree bias (§4.3); and b) early termination and not jumping back to trust seeds (§4.2.1).

Impact of the connectivity of the Sybil region. We examine how the connection density within the Sybil region impacts the node ranking generated by EigenTrust and SybilRank. To do so, we vary the number of edges each Sybil has with other Sybils from 4 to 40 under a regular attack on the Facebook graph. We refer to such edges as *non-attack*.

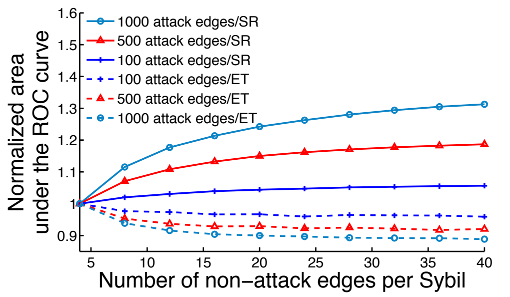
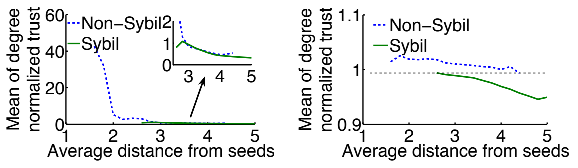


Figure 7: Normalized area under the ROC curve as a function of the number of edges connecting to other Sybils.

Figure 7 shows the *normalized area under the ROC curve*, which is computed by dividing the area under the ROC curve for each attack scenario with the baseline case where each Sybil has only 4 non-attack edges. As can be seen in Figure 7, with EigenTrust, the value of the area under the ROC curve decreases when the connections within the Sybil region become dense (the normalized area under the ROC curve is always less than 1). This result is because in EigenTrust a node that has many incoming links or is pointed to by other highly-ranked nodes is typically ranked high [33, 45]. A malicious user can simply create dense connections within the Sybil region to boost the ranks of selected Sybils. Therefore, denser Sybil connections lower EigenTrust's values of the area under the ROC curve, indicating that more Sybils are ranked higher than non-Sybils.

On the contrary with SybilRank, due to the removal of degree bias, high-degree Sybils do not benefit. Instead, SybilRank's performance improves as the connections among Sybils become denser. This is because it exploits the sparse cut between the Sybil and the non-Sybil region, which the addition of Sybil connections makes even sparser (it lowers the conductance).

Impact of the distance from trust seeds. As we discuss in §4.2.1, EigenTrust's trust distribution is sensitive to the locations of the seeds because its random walks



(a) EigenTrust (b) SybilRank

Figure 8: Trust distribution with respect to the average distance to seeds in the synthetic graph under the regular attack with 10000 **attack edges.**

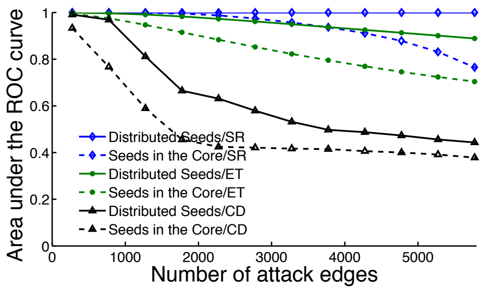


Figure 9: Comparing SybilRank, EigenTrust and Mislove's CD in a multi-community graph.

jump back to them. Figure 8 compares the distribution of *degree-normalized* trust generated by EigenTrust and SybilRank with respect to a node's average shortest hop distance from the trust seeds. We simulate a regular attack in the synthetic graph with $10K$ attack edges, and select 5 seeds for both EigenTrust and SybilRank. The total trust is set to $2m$. As shown in Figure 8(a), EigenTrust tends to allocate high degree-normalized trust to nodes close to the seeds. Nodes relatively farther from the seeds get substantially lower trust. In fact, the degree-normalized trust distribution in EigenTrust has a "heavy" tail. Among the $10K$ non-Sybil nodes, more than $9.4K$ have degree-normalized trust lower than 2. As we zoom into the tail, this large set of non-Sybil users and Sybils have indistinguishable degree-normalized trust .

Figure 8(b) shows that unlike EigenTrust, SybilRank assigns roughly the same degree-normalized trust to each non-Sybil node, while keeping a distinguishable gap between non-Sybils and Sybils. This empirically validates our observation that the degree-normalized trust is approximately uniformly distributed in the non-Sybil region after $O(\log n)$ power iterations (§4.3).

6.4 Coping with multiple communities

As discussed in §4.2.2, we distribute seeds among communities to cope with the multi-community structure in the non-Sybil region. We illustrate this with simulations on synthetic graphs. The simulations also include Mislove's CD and EigenTrust, which can be initialized with multiple seeds at no additional computational cost. We do not include GateKeeper because it uses random walks to seek seeds (ticket sources), thus we do not fully control the placement of seeds among the communities.

Similar to the simulation scenarios in [53], we set up a non-Sybil region consisting of 5 scale-free synthetic communities, each of which has 2000 nodes with an average degree of 10. We designate one community as the core of the social graph. The other 4 communities do not have any connections to each other, but connect to the graph core via only 500 edges. This process builds a non-Sybil region where multiple communities connect to the core community of the graph via limited links.

We contrast the following two seed selection strategies: a) all 50 seeds are confined to the core community; and b) the seeds are distributed among all the non-Sybil communities, each of which has 10 seeds. As shown in Figure 9, seed selection strategy (b) improves the detection accuracy for all schemes. Wide seed coverage in the non-Sybil region offsets the impact of its multi-community structure: the resulting ranking is mostly determined by the sparse cut between the non-Sybil region and the Sybil region. In Figure 9, we see that SybilRank maintains the highest accuracy for each of the seed placement strategies due to the disadvantages of EigenTrust and Mislove's CD mentioned in §6.3 and §6.2.

6.5 Resilience to seed-targeting attacks

Last, we study how various schemes perform under targeted attacks. In these attacks, sophisticated attackers may obtain partial or full knowledge about the OSN and discover the locations of the trust seeds. They can then establish attack edges to nodes close to the seeds.

We simulate the targeted attack in the Facebook graph. The 5K-node Sybil region forms a regular attack structure, and has 200 attack edges connecting to the non-Sybil region. Instead of being completely randomly distributed, those 200 attack edges are established by randomly connecting to the k non-Sybil nodes with the shortest distance from the trust seed. For schemes supporting multiple seeds, we target the attack edges to the highest-degree trust seed. We vary k from 1K to 10K. A smaller k signifies a shorter average distance between Sybils and seeds.

As shown in Figure 10, when attack edges are attached close to the seed, all schemes' performance degrades, while SybilRank keeps the most stable performance across a wide range of k values. This is because SybilRank does not assign excessive trust to nodes that are close to the seeds (§6.3). However, SybilRank's performance still degrades when k is small. This is because these closely targeted attacks force SybilRank to "leak" a fraction of trust in the Sybil region during early power iterations. Thus, its detection accuracy reduces as Sybils may gain higher trust than non-Sybils.

7 Real-world Deployment

We now discuss the deployment of our system on a snapshot of Tuenti's complete social friendship graph, which

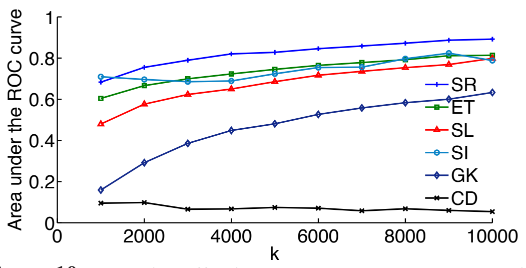


Figure 10: **Detection effectiveness under the attacks targeting the k non-Sybils with the shortest distance to the seed.**

was obtained in August 2011. Due to the sheer volume of users, it would be infeasible for us to manually inspect whether each user is a Sybil. Thus, we are unable to evaluate SybilRank with the same metrics as in the simulations (§6), such as the area under the ROC curve, the false negative rate and the false positive rate. Instead, we attempt to determine the portion of fake users at varying segments of the ranked list. We do so by manually inspecting a user sample in each particular interval in the ranked list (§4.4). Due to the practical constraints and the limited availability of human verifiers, we do not deploy the other Sybil defenses on Tuenti.

Pre-processing. We observe that in Tuenti some fake Tuenti accounts are well-maintained and have extremely high degree. They may introduce many attack edges if they connect to real users. At the same time, brand new real users always have weak connectivity to others due to the limited time they have been in the OSN, resulting in false positives. To reduce the impact of these two factors, we perform pre-processing before applying SybilRank: we prune the edges on extremely-high-degree nodes and defer the consideration of very recent users (see [20]).

Communities in Tuenti. The complete Tuenti social graph has 1,421,367,504 edges and 11,291,486 nodes, among which 11,216,357 nodes form a Giant Connected Component (GCC). Our analysis focuses on the GCC. With the Louvain method, we found 595 communities, among which 25 large communities contain more than 100K nodes. We inspected 4 nodes in each community and designated as SybilRank trust seeds the nodes that pass the manual verification.

7.1 Validating the mixing time assumptions

As discussed in §3.2, SybilRank relies on the mixing time gap between the non-Sybil region and the entire graph. Since we seed trust in each large community, the trust propagation is mainly determined by those communities. To this end, we investigate the 25 large communities identified by the Louvain method. As shown in §7.2, fake accounts are embedded in each community. We then measure the mixing time gap between each community and its non-Sybil part.

We measure the mixing time using its definition, i.e., the maximum necessary walk length to achieve a given

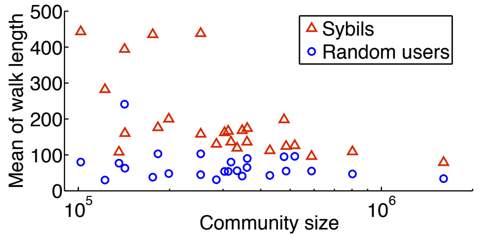


Figure 11: **Contrasting the mean length of random walks from Sybils and from random users in each community.**

variation distance [39] from the stationary distribution. Due to Tuenti's large population, it is difficult to remove all Sybils in order to measure the mixing time of the non-Sybil part in each community. Therefore, our measurement approximates the mixing time gap. We do so by contrasting the mean length of random walks from random users and from the Sybils that are captured by SybilRank in each community.

We hypothesize that due to the majority of users in Tuenti being non-Sybil, if the Sybils are weakly connected to the majority, given a total variation distance, the random walks from Sybils need to be longer than that from average users. In such a case, the Sybils' random walk length can approximate the mixing time of the entire community, and we designate the length of random walks from average users as the mixing time of the non-Sybil part. Since the average users may include hidden Sybils, using their walk length only overestimates the mixing time of the non-Sybil part.

We select 1000 random users and 100 confirmed Sybils in each community. The total variation distance is set equal to 0.01. As shown in Figure 11, the mean length of random walks from the random users in each community is small (mostly < 100), while the mean Sybil walk length is much longer. This indicates that the Sybils connect to the majority users with a limited number of edges, which makes their needed walk length longer. This fact demonstrates that social-graph-based defenses can be effective for our real OSN graph. In addition, recall that in SybilRank we have placed multiple random seeds in each community, which facilitates the convergence of the trust propagation. The power iterations that SybilRank needs are even less than the random walk length in Figure 11.

7.2 Detecting Fakes in Tuenti

Manually inspecting the ranked list. We run SybilRank on the complete Tuenti social graph. We inspected 2K users at the bottom of the resulting ranked list, and all of them were fake (Sybils). We further examined the ranked list by inspecting the first lowest-ranked one million users. We randomly selected 100 users out of each 50K-user interval for inspection. As shown in Figure 12, the 100% fake portion was maintained at the first 50K-user interval, but the fake portion gradually decreased as we went up the list. Up to the first 200K lowest-ranked

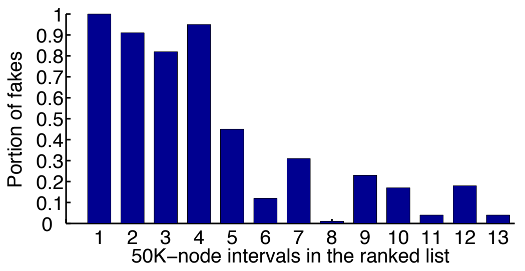


Figure 12: **Sybil distribution over the lowest 650K users on the ranked list (intervals are numbered from the bottom).**

users, around 90% are fake, as opposed to the ∼5% hit rate of Tuenti's current abuse-report-based method. This suggests an 18-fold increase in the efficiency with which Tuenti can process suspected accounts.

We also observe that above the 200K lowest-ranked users, the portion of fakes decreases abruptly from ∼90% to ∼50%, and then ∼10%. This is because fake accounts that have established many social links to real users can be ranked high. This reveals that SybilRank's limitation lies in the open nature of OSNs, i.e., the ease at which some fake accounts can befriend real users.

Although we have sampled users for inspection until the lowest 1 million users, due to confidentiality reasons we report the exact portions of fake users until the lowest 650K users. Above the lowest 650K, we obtain even lower portions of fakes, which subsequently stabilize. One can sample above the point in which the portion of fakes stabilizes to infer the portion of fakes in the complete Tuenti network. We have also obtained a more accurate estimate of the portion of fakes in the network by uniformly sampling over the complete network. This allows us to determine how many fake accounts are not captured by SybilRank. Again, we cannot reveal this statistic, as we are bound by confidentiality.

The portion of fakes we report is directly related to *precision* [32], which is a performance metric used in Collaborative Filtering. It is the ratio of relevant items over the top θ highest ranked items in terms of relevance. The results in Figure 12 reflect a precision as high as 90% among the first 200K lowest-ranked users (see [20]).

Formation of the Sybil collective. We showed above that SybilRank achieves an almost 100% portion of fakes in the first $50K$ lowest-ranked users. We now study the social connections among those $50K$ users. We found that fakes in Tuenti are rarely isolated from each other and that real world Sybils exhibit various formations (see [20]). We observed three large connected components that manifest a simple tree-like connection pattern between nodes of similar degree. This indicates that those accounts may be automatically crafted for spam, rating manipulation, and other attacks on a large scale.

In addition, those $50K$ users do not form a single connected component. Instead they form many separate connected clusters. Presumably this is due to the fact that the attackers behind those fake accounts are not centrally coordinated. We also investigate the degree of those $50K$ users. 80% of these users have no more than 10 friends, while there are hundreds among these Sybils that have more than 100 friends. This indicates that the node degree is not a reliable metric to detect fake accounts due to its large variance among fake users.

Sybils in large communities. SybilRank leverages the community structure to properly seed trust. It is much more accurate than just determining if an entire community is fake. Among the $50K$ lowest-ranked users, we found that part of them are embedded in the 25 large communities (see [20]). For example, each top-10 largest community has hundreds of Sybils. This indicates that SybilRank detects fake accounts even in large communities that mostly consist of non-Sybil users.

7.3 Discussion

With SybilRank, Tuenti needs ∼570 man hours to go over the 200K lowest ranked users and discover ∼180K fake accounts. With its current abuse-report-based method, which has only ∼5% hit rate, and assuming all these fakes are reported, Tuenti would need ∼10,300 hours. By executing SybilRank periodically, e.g., every month, we expect Tuenti to remain able to efficiently identify a substantial number of fake accounts.

Our study pin-pointed 200K suspicious accounts after a total of 20 hours of SybilRank and community estimation processing. Those accounts can be verified by one full-time (8h) employee in ∼70 days. This compares favorably to the feature-based detection mechanism used by Yang et al. [54], which unveiled in real time 100K fakes in the 120M-user RenRen, over 6 months.

Yang et al. [54] reported that 70% of their detected fake nodes do not have any connections to other fakes. However, unlike RenRen, Tuenti is invitation-only. Thus, a fake account is at least connected to its inviter when created. As a result, the number of isolated accounts is small (<70K) compared to the number of the fake accounts detectable by SybilRank. In addition, we found that Sybils in Tuenti do form dense connections among themselves, which as we show in §6.3 further enables SybilRank to uncover Sybils. Last, we note that most detected fake accounts were spammers [12].

8 Conclusion

Large scale social online services place immense attention to the experience of their user base, and the marketability of their user profiles and the social graph. In this context, they face a significant challenge by the existence and continuous creation of fake user accounts, which dilutes the advertising value of their network and annoys legitimate users. To this end, we have proposed

SybilRank, an effective and efficient fake account inference scheme, which allows OSNs to rank accounts according to their perceived likelihood of being fake. Therefore, this work represents a significant step towards practical Sybil defense: it enables an OSN to focus its expensive manual inspection efforts, as well as to correctly target existing countermeasures, such as CAPTCHAs.

9 Acknowledgments

We are grateful to the anonymous reviewers, our shepherd John Byers, Ang Li, and Vijay Erramilli for their valuable feedback. We are particularly thankful to Konstantina Papagiannaki for her extensive feedback on our paper's presentation. We also thank Nguyen Tran, Jinyang Li, Alan Mislove, and George Danezis for making their source code available for our evaluation. This work was partly funded by NSF Awards CNS-0845858 and CNS-1017858.

References

[1] Apache Hadoop. http://hadoop.apache.org/.
[2] Facebook connect. developers.facebook.com/connect.php.
[3] Stanford Large Network Dataset Collection. http://snap.stanford.edu/data/index.html.
[4] Tuenti: A Private, Invitation-only Social Platform Used by Millions of People to Communicate and Share Every Day. http://www.tuenti.com.
[5] An Explanation of CAPTCHAs. http://www.facebook.com/note.php?note_id=36280205765, 2008.
[6] Fake Accounts in Facebook - How to Counter it. http://tinyurl.com/5w6un9u, 2010.
[7] Stolen Facebook Accounts for Sale. http://tinyurl.com/25cngas, 2010.
[8] Tuenti, Spain's Leading Social Network, Switches on Local for a Location-based Future. http://tinyurl.com/yeygmw5, 2010.
[9] Why the Number of People Creating Fake Accounts and Using Second Identity on Facebook are Increasing. http://tinyurl.com/3uwq75x, 2010.
[10] Google+ Account Suspensions Over ToS Drawing Fire. http://tinyurl.com/5vrt524, 2011.
[11] Google Explores +1 Button To Influence Search Results. http://tinyurl.com/7g927oy, 2011.
[12] Personal communication with the Manager of User Support and the Product Manager of the Core and Community Management teams in Tuenti, 2011.
[13] The Facebook Blog: A Continous Commitment to Security. https://www.facebook.com/blog/blog.php?post=486790652130, 2011.
[14] L. V. Ahn, M. Blum, N. J. Hopper, and J. Langford. CAPTCHA: Using Hard AI Problems for Security. In *Eurocrypt*, 2003.
[15] A.-L. Bárabási and R. Albert. Emergence of Scaling in Random Networks. *Science*, 286:509–512, 1999.
[16] E. Behrends. Introduction to Markov chains: with Special Emphasis on Rapid Mixing. In *Advanced Lectures in Mathematics*, 2000.
[17] L. Bilge, T. Strufe, D. Balzarotti, and E. Kirda. All Your Contacts Are Belong to Us: Automated Identity Theft Attacks on Social Networks. In *WWW*, 2009.
[18] V. D. Blondel, J.-L. Guillaume, R. Lambiotte, and E. Lefebvre. Fast Unfolding of Communities in Large Networks. In *Journal of Statistical Mechanics: Theory and Experiment*, 2008.
[19] Y. Boshmaf, I. Muslukhov, K. Beznosov, and M. Ripeanu. The Socialbot Network: When Bots Socialize for Fame and Money. In *ACSAC*, 2011.
[20] Q. Cao, M. Sirivianos, X. Yang, and T. Pregueiro. Aiding the Detection of Fake Accounts in Large Scale Social Online Services. Technical Report, http://www.cs.duke.edu/~qiangcao/publications/sybilrank_tr.pdf, 2011.
[21] A. Cheng and E. Friedman. Sybilproof Reputation Mechanisms. In *P2PEcon*, 2005.
[22] P.-A. Chirita, J. Diederich, and W. Nejdl. MailRank: Using Ranking for Spam Detection. In *CIKM*, 2005.
[23] G. Danezis and P. Mittal. SybilInfer: Detecting Sybil Nodes Using Social Networks. In *NDSS*, 2009.
[24] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*, 2004.
[25] M. Dell amico and Y. Roudier. A Measurement of Mixing Time in Social Networks. In *5th International Workshop on Security and Trust Management*, 2009.
[26] J. R. Douceur. The Sybil Attack. In *IPTPS*, 2002.
[27] P. W. L. Fong. Preventing Sybil Attacks by Privilege Attenuation: A Design Principle for Social Network Systems. In *IEEE S&P*, 2011.
[28] H. Gao, J. Hu, C. Wilson, Z. Li, Y. Chen, and B. Y. Zhao. Detecting and Characterizing Social Spam Campaigns. In *IMC*, 2010.
[29] M. Gjoka, M. Kurant, C. T. Butts, and A. Markopoulou. A Walk in Facebook: Uniform Sampling of Users in Online Social Networks. In *IEEE INFOCOM*, 2010.
[30] Z. Gyöngyi, H. Garcia-Molina, and J. Pedersen. Combating Web Spam with TrustRank. In *VLDB*, 2004.
[31] J. A. Hanley and B. J. McNeil. The Meaning and Use of the Area under a Receiver Operating Characteristic (ROC) Curve. *Radiology*, 143, 1982.
[32] J. L. Herlocker, J. A. Konstan, L. G. Terveen, and J. T. Riedl. Evaluating Collaborative Filtering Recommender Systems. *ACM Trans. Inf. Syst.*, 22, 2004.
[33] S. D. Kamvar, M. T. Schlosser, and H. Garcia-Molina. The EigenTrust Algorithm for Reputation Management in P2P Networks. In *WWW*, 2003.
[34] A. N. Langville and C. D. Meyer. Deeper Inside Pagerank. *Internet Mathematics*, 1:2004, 2004.
[35] J. Leskovec and C. Faloutsos. Sampling from Large Graphs. In *ACM SIGKDD*, 2006.
[36] C. Lesniewski-Laas and M. F. Kaashoek. Whanau: A Sybil-proof Distributed Hash Table. In *NSDI*, 2010.
[37] A. Mislove, A. Post, P. Druschel, and K. P. Gummadi. Ostra: Leveraging Social Networks to Thwart Unwanted Traffic. In *NSDI*, 2008.
[38] A. Mislove, B. Viswanath, K. P. Gummadi, and P. Druschel. You are Who You Know: Inferring User Profiles in Online Social Networks. In *ACM WSDM*, 2010.
[39] M. Mitzenmacher and E. Upfal. *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*. Cambridge University Press, 2005.
[40] A. Mohaisen, N. Hopper, and Y. Kim. Keep Your Friends Close: Incorporating Trust into Social-Network-based Sybil Defenses. In *INFOCOM*, 2011.
[41] A. Mohaisen, A. Yun, and Y. Kim. Measuring the Mixing Time of Social Graphs. In *ACM IMC*, 2010.
[42] M. Mondal, B. Viswanath, A. Clement, P. Druschel, K. P. Gummadi, A. Mislove, and A. Post. Limiting Large-scale Crawls of Social Networking Sites. In *SIGCOMM Poster Session*, 2011.
[43] M. Motoyama, D. McCoy, K. Levchenko, G. M. Voelker, and S. Savage. Dirty Jobs: The Role of Freelance Labor in Web Service Abuse. In *Proceedings of the USENIX Security Symposium*, 2011.
[44] A. Nazir, S. Raza, C.-N. Chuah, and B. Schipper. Ghostbusting Facebook: Detecting and Characterizing Phantom Profiles in Online Social Gaming Applications. In *WOSN*, 2010.
[45] L. Page, S. Brin, R. Motwani, and T. Winograd. The PageRank Citation Ranking: Bringing Order to the Web. Technical report, Stanford InfoLab, 1999.
[46] A. Post, V. Shah, and A. Mislove. Bazaar: Strengthening User Reputations in Online Marketplaces. In *USENIX NSDI*, 2011.
[47] D. Quercia and S. Hailes. Sybil Attacks Against Mobile Users: Friends and Foes to the Rescue. In *INFOCOM*, 2010.
[48] P. Resnick and R. Sami. Sybilproof Transitive Trust Protocols. In *ACM Electronic Commerce Conference*, 2009.
[49] R. Sommer and V. Paxson. Outside the closed world: On using machine learning for network intrusion detection. In *IEEE S&P*, 2010.
[50] K. Thomas, C. Grier, V. Paxson, and D. Song. Suspended Accounts in Retrospect: An Analysis of Twitter Spam. In *IMC*, 2011.
[51] D. N. Tran, B. Min, J. Li, and L. Subramanian. Sybil-Resilient Online Content Rating. In *NSDI*, 2009.
[52] N. Tran, J. Li, L. Subramanian, and S. S. Chow. Optimal Sybil-resilient Node Admission Control. In *INFOCOM*, 2011.
[53] B. Viswanath, A. Post, K. P. Gummadi, and A. Mislove. An Analysis of Social Network-based Sybil Defenses. In *ACM SIGCOMM*, 2010.
[54] Z. Yang, C. Wilson, X. Wang, T. Gao, B. Y. Zhao, and Y. Dai. Uncovering Social Network Sybils in the Wild. In *IMC*, 2011.
[55] H. Yu. Sybil Defenses via Social Networks: A Tutorial and Survey. In *ACM SIGACT News, Distributed Computing Column*, 2011.
[56] H. Yu, P. Gibbons, M. Kaminsky, and F. Xiao. SybilLimit: A Near-Optimal Social Network Defense Against Sybil Attacks. In *IEEE S&P*, 2008.
[57] H. Yu, M. Kaminsky, P. B. Gibbons, and A. Flaxman. SybilGuard: Defending Against Sybil Attacks via Social Networks. In *SIGCOMM*, 2006.
[58] H. Yu, C. Shi, M. Kaminsky, P. B. Gibbons, and F. Xiao. DSybil: Optimal Sybil-Resistance for Recommendation Systems. In *IEEE S&P*, 2009.
[59] Y. Zhang, J. Wang, Y. Wang, and L. Zhou. Parallel Community Detection on Large Networks with Propinquity Dynamics. KDD, 2009.
[60] Y. Zhao, Y. Xie, F. Yu, Q. Ke, Y. Yu, Y. Chen, and E. Gillum. BotGraph: Large Scale Spamming Botnet Detection. In *NSDI*, 2009.

Don't Lose Sleep Over Availability: The GreenUp Decentralized Wakeup Service

Siddhartha Sen[†], Jacob R. Lorch, Richard Hughes, Carlos Garcia Jurado Suarez,
Brian Zill, Weverton Cordeiro[‡], and Jitendra Padhye
Microsoft Research [†]*Princeton University* [‡]*Universidade Federal do Rio Grande do Sul*

Abstract

Large enterprises can save significant energy and money by putting idle desktop machines to sleep. Many systems that let desktops sleep and wake them on demand have been proposed, but enterprise IT departments refuse to deploy them because they require special hardware, disruptive virtualization technology, or dedicated per-subnet proxies, none of which are cost-effective. In response, we devised GreenUp, a minimal software-only system that allows any machine to act as a proxy for other sleeping machines in its subnet. To achieve this, GreenUp uses novel distributed techniques that spread load through randomization, efficiently synchronize state within a subnet, and maintain a minimum number of proxies despite the potential for correlated sleep times. In this paper, we present the details of GreenUp's design as well as a theoretical analysis demonstrating its correctness and efficiency, using empirically-derived models where appropriate. We also present results and lessons from a seven-month live deployment on over 100 machines; a larger deployment on ~1,100 machines is currently ongoing.

1 Introduction

A number of recent studies [2, 4, 18, 32] show that desktop computers in enterprise environments collectively waste a lot of energy by remaining on even when idle. By putting these machines to sleep, large enterprises stand to save millions of dollars [31].

The reasons machines stay awake when idle are well known [2, 23]. Most OSes put a desktop machine to sleep after some amount of user idle time, but users and IT administrators override this to enable remote access "at will." In enterprise environments, users typically access files or other resources from their machines, while IT administrators access machines for maintenance tasks. This scenario is prevalent in Microsoft and other large corporations [16]. Thus, any system for putting machines to sleep must maintain their availability.

A number of solutions have been proposed to solve this problem [2, 3, 8, 18]. However, many of the proposed solutions are difficult to deploy. For example, Somniloquy [2] requires specialized hardware, Litegreen [8] requires a fully virtualized desktop, and SleepServer [3] requires special application stubs.

The most promising approach, which requires no hardware changes and only minimal software deployment, is the "sleep proxy" approach, first proposed by Allman et al. [4] and further detailed by others [18, 23]. The key idea is that traffic meant for each sleeping machine is directed to a proxy. The proxy inspects the traffic, answers some such as ARP requests on behalf of the sleeping machine, and wakes the machine when it sees important traffic such as a TCP SYN. The proxy discards most of the traffic without any ill effects [18].

We previously deployed such an approach [23], but when we tried to start large-scale trials of it, we ran into major difficulties with our IT department that required us to completely redesign our solution. Basically, they refused to deploy and maintain a dedicated proxy on every subnet, due to the costs involved. Moreover, such a proxy constitutes a single point of failure, and would necessitate additional backup proxies, further adding to the cost. These opinions were also shared by the IT departments of some of Microsoft's large customers. We discuss this more in §2.

These considerations led us to a decentralized design for GreenUp, our system for providing availability to machines even as they sleep. The key idea behind GreenUp is that any machine can act as a sleep proxy for one or more sleeping machines on the same subnet. Whenever a machine falls asleep, another one starts acting as a sleep proxy for it. If the sleep proxy itself falls asleep, another sleep proxy rapidly takes over its duties.

Like most distributed systems, GreenUp must handle issues of coordination and availability among participants that operate independently. However, our problem setting has several distinctions that led us to an atypical design. First, machines are on the same subnet, so they can efficiently broadcast to each other and wake each other up. Second, GreenUp does not require a strongly-consistent view of the system. Third, GreenUp runs on end-user machines, making them sensitive to load and inherently unreliable. That is, their users will not tolerate noticeable performance degradation, and they may go to sleep at any time.

The techniques we have developed for this environment can be applied to any distributed system facing similar types of machine behavior and consistency requirements. *Distributed management* uses randomization to spread sleep proxy duties evenly over awake machines without explicit coordination. *Subnet state coordination* uses the subnet broadcast channel shared by all participants to efficiently disseminate state among them. Additionally, we show how *guardians* can protect against a new type of correlated failure we observe: correlated sleep times among participants.

Our work makes the following contributions:

- We provide a complete design for a distributed system that makes sleeping machines highly available, including novel techniques for distributed management, subnet state coordination, and mitigation of correlated sleep with guardians (§4).
- We analytically justify our design techniques and prove that they achieve the properties we require. Where appropriate, we use models derived from real sleep behavior (§5).
- We demonstrate the feasibility, usability, and efficiency of our design by implementing it and deploying it on over 100 users' machines (§6), and discuss the many lessons we have learned from this deployment over the past seven months (§7).

After providing motivation in §2 and background useful for understanding our design in §3, §4–§7 detail our contributions. §8 discusses key issues and areas of ongoing work, §9 presents related work, and §10 concludes.

2 Motivation

In our earlier work [23], we discussed the engineering issues involved in deploying a sleep proxy solution. However, when we wanted to start large-scale trials of our system in collaboration with Microsoft's IT department (MSIT), we ran into difficulties that required us to completely re-engineer it.

MSIT was eager to deploy a solution that let idle desktops sleep while keeping them accessible. Currently, MSIT mandates and enforces a common sleep policy that achieves energy savings but reduces availability: user desktops automatically sleep after 30 minutes of inactivity. The goal is not just to save money, but also to contribute to Microsoft's overall environmental impact reduction initiative. If individual users wish to exclude their machines from this mandate, they have to request exceptions for each individual machine, and must provide a reason for seeking the exception. Based on an informal analysis of the reasons mentioned by the users seeking exceptions, MSIT decided that a solution that would automatically wake a sleeping desktop upon remote access would reduce exception requests.

Since deploying special hardware [2] or full virtualization [8] would be both too disruptive and too costly [23], these were not feasible, and only two options remained.

One option was to use the ARP proxy functionality offered by many modern NICs, combined with their ability to trigger wakeup when observing packets with specific bit patterns. However, this approach fails in the presence of complex encapsulation and IPSec encryption, both of which are commonly used in enterprise networks [23]. We also found that wake-on-pattern functionality in many modern NICs is unreliable.

The other possibility was to use a sleep proxy system [23], but this too had problems. One issue was deployment cost, due to its requirement of a dedicated machine on each subnet. Microsoft's corporate network consists of hundreds of subnets, so the cost of deploying and, more importantly, managing dedicated servers on each subnet would have been too high.

Worse yet, these servers would constitute a new single point of failure for each subnet, impacting the robustness and availability of the system. Our user studies showed that most users typically do not access their machines remotely, but when they do, the need is usually critical. For example, a salesperson may want to access some documents from his desktop while meeting with a client. Thus, each subnet would require additional backup servers and a system for monitoring liveness to achieve high availability, further raising the cost.

The same concerns were echoed by IT departments of some of Microsoft's largest customers [16]. As a result, a thorough re-thinking of the solution was needed. In particular, we decided to re-architect the system to make it completely distributed and serverless.

3 Background

This section provides a brief primer on Wake-on-LAN technology and on the original sleep proxy design [18, 23]. The discussion is not comprehensive; it only covers details useful in understanding our design.

3.1 Wake-on-LAN technology

Basic Wake-on-LAN (WoL) technology [5] has been available in Ethernet NICs for quite some time. To use it, a machine enters the S3 or S5 sleep state [11] while the NIC stays powered on. If the NIC receives a special *magic packet*, it wakes the machine. The magic packet is an Ethernet packet sent to the broadcast address whose payload contains 16 consecutive repetitions of the NIC's MAC address [5].

Because the packet is sent to the broadcast address, it is generally not possible to wake a machine on a different subnet. Some NICs do respond to WoL packets sent to unicast addresses, but this is not true for all NICs. Subnet directed broadcasts could wake machines in other subnets, but only with special support from routers. Thus, for robustness, our system relies only on basic WoL functionality and operates only within a single subnet.

3.2 Basic design of the sleep proxy

The core sleep proxy functionality of our system is similar to that described by Reich et al. [23] We summarize it via an example. Consider two machines S and P, both on the same subnet. S will fall asleep, and P is the dedicated sleep proxy for the subnet.

When the operating system on S decides the machine should sleep, it sends a "prepare to sleep" notification to all running processes. A daemon running on S waits for this event, then broadcasts a packet announcing that S is about to sleep. The packet contains S's MAC and IP addresses, and a list of open TCP ports in the listen state.

When P receives this packet, it waits for S to fall asleep. Once pings indicate S has fallen asleep, P sends appropriately crafted ARP probes binding S's IP address to S's MAC address [23]. These trigger the underlying layer-2 spanning tree to be reprogrammed such that all packets meant for S are delivered to P instead. Notice that this is *not* address hijacking: it is *Ethernet port hijacking* [23], which is more general and robust against various DHCP issues.

P operates its NIC in promiscuous mode, and hence can observe packets meant for S. P ignores most traffic meant for S [18], with two exceptions.

First, P answers ARP requests and IPv6 neighbor-solicitation messages directed to S. This "ARP offload" functionality is needed to ensure other machines can direct packets to S's address [18].

Second, if P observes a TCP SYN directed to an open port on S, it wakes S by broadcasting a magic packet containing S's MAC address. The OS on S, upon waking up, will send its own ARP probes. These reprogram the layer-2 spanning tree so that all subsequent packets meant for S are correctly delivered to S instead of to P.

Meanwhile, the client that sent the original TCP SYN retransmits the SYN several times: 3 sec later, 6 sec after that, 12 sec after that, etc. Eventually, a retransmitted SYN reaches S, which is now awake and can complete the connection in a normal manner. The user experiences a small delay since the first SYN is not responded to [23], but otherwise the user experience is seamless.

4 GreenUp Design

The primary goal of GreenUp is to ensure that all participants remain highly available while allowing them to sleep as much as possible. A secondary goal is to rely only on participant machines, not dedicated servers, to aid availability and ease deployment. Designing such a system presents unique opportunities and challenges:

(i) Subnet domains. Each GreenUp instance runs within a subnet domain, providing an efficient broadcast channel for communicating and for waking machines up.

(ii) Availability over consistency. The availability of machines is more important than the consistency of system state or operations. In particular, strong consistency can be sacrificed in favor of simplicity or efficiency.

(iii) Load-sensitive, unreliable machines. Running on end-user machines requires us to limit the resources we use, *e.g.* CPU cycles, since users will not tolerate noticeable performance degradation. Also, these machines may go to sleep at any time, and they may exhibit correlated behavior, *e.g.* sleeping at similar times.

GreenUp's design exploits these opportunities and meets these challenges, using novel techniques that apply to a broader class of distributed systems. GreenUp is suitable for enterprises where desktop machines provide services to remote users. This is the case for Microsoft and some of its large customers, as we have discussed. In organizations where desktops are used only as terminals, or where users only use mobile laptops, GreenUp is not appropriate. In §8, we discuss possible extensions of GreenUp for certain server cluster settings.

4.1 Overview

To aid the presentation of our design, we introduce the following terminology. We call a machine with GreenUp installed a *participant*. When a participant is capable of intercepting traffic to sleeping machines, we call it a *proxy*. Generally, every participant is a proxy as long as it is awake. When a proxy is actively intercepting traffic for one or more sleeping machines, we say it is *managing* them: it is their *manager* and they are its *managees*. To manage a participant, a manager needs to know the managee's *state*: its IP address, its MAC address, and its list of open TCP ports.

The key idea for maintaining availability in GreenUp is to enable any proxy to manage any sleeping participant on the same subnet. When a participant goes to sleep, some proxy rapidly detects this and starts managing it. For this, we use a scheme called *distributed management* that spreads management load evenly over awake proxies by using randomized probing (§4.2). To detect and manage asleep machines, each proxy must know each participant's state, so we distribute it using a scheme based on subnet broadcast called *subnet state coordination* (§4.3). This scheme tolerates unreliable machines and provides eventual consistency for participants' views of state. Distributed management will stop working if all proxies go to sleep, a condition we call *apocalypse*. Using real trace data, we demonstrate that such correlated sleep behavior is plausible, and consequently show how to use *guardians* (§4.4) to always maintain a minimum number of awake proxies.

Figure 1 shows an example illustration of these techniques from our live deployment. The coming sections explain them in greater detail.

4.2 Distributed management

In this section, we discuss our scheme for *distributed management*. The techniques we develop here can be used by any distributed system to coordinate a global task using local methods, provided the system can tolerate occasional conflicts. In our scheme, proxies do not explic-

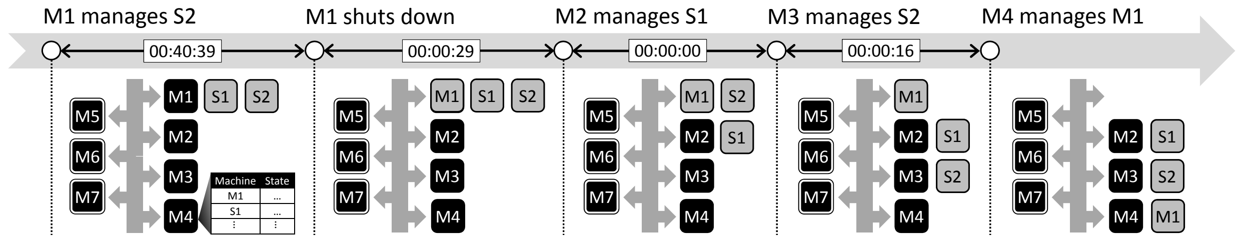


Figure 1: Live operation of GreenUp. Awake proxies are colored black, guardians have an additional white border, and asleep or shut down machines are gray. Participants use subnet state coordination to exchange state, shown in the inset of M4. M1 starts shutting down at 11:18 am on July 11; its abandoned managees S1 and S2 are managed again 29 seconds later via distributed management. M1 is managed 16 seconds after that, since it required time to shut down. Guardians M5, M6, and M7 stay awake to prevent apocalypse.

itly coordinate, yet they ensure a sleeping participant is rapidly detected and managed (§4.2.1). In the event that multiple proxies accidentally manage the same participant, they quickly detect this and all but one back off (§4.2.2). Due to our distributed setting, our managers operate differently (§4.2.3) from the sleep proxy described in §3.2.

4.2.1 Distributed probing

To maintain the availability of sleeping machines, we need to ensure that asleep participants are always managed. It might seem that we could rely on participants to broadcast a notification when they are about to sleep, but this is unreliable. The notification may not be seen if there is network loss, if the participant fails, or if the participant is already asleep and its current manager fails. Thus, proxies must continually monitor participants to discover when they become both asleep and unmanaged.

As we will see in §7.2, monitoring every participant on a large subnet can cause noticeable load. Because of the load sensitivity challenge, we distribute the monitoring work across all proxies via *distributed probing*, as follows. Each proxy periodically *probes* a random subset of participants. If a participant is awake, it responds to the probe; otherwise, its manager responds on its behalf. If the proxy receives no response, it starts managing the machine itself. Note that assigning the management task to the proxy that first detects the sleeping machine effectively distributes the management load across all proxies, as we prove in §5.2.

A proxy *probes* a participant by sending it two packets: a TCP SYN to a GreenUp-specific port, and an ICMP echo request or *ping*. If the machine is awake, its operating system will send an ICMP echo reply. If the machine is managed, its manager will be watching for TCP SYNs to it, and will respond to the probe with a UDP message indicating the machine is managed. If the proxy hears neither response, it resends the probe packets every second until a sufficiently long period passes without a response, at which point it concludes the participant is asleep and unmanaged. In our current implementation, we use a timeout of 25 seconds, which we consider a reasonable amount of time to ascertain a machine is asleep.

Our goal is to start managing participants quickly once they or their current managers fall asleep. This means we cannot let a participant go very long without some proxy probing it, necessitating a periodic rather than exponential-backoff scheme. Thus, each proxy uses the following random probing schedule to ensure each participant is probed once every s seconds with probability p. Given $\mathcal{S}$, the set of participants it is not managing, and $\mathcal{K}$, the set of proxies, it probes a random subset of $\mathcal{S}$ of size $-\ln(1-p)\,|\mathcal{S}|/|\mathcal{K}|$ every s seconds. We prove in §5.1 that this quantity achieves our goal. In our current implementation, we use $p = 90\%$ and $s = 3$, so an asleep unmanaged machine is probed within 3 seconds with probability 90% and 6 seconds with probability 99%. These parameters limit the additional delay beyond 25 seconds that we must wait to ensure a machine is asleep.

4.2.2 Resolving multiple management

Distributing responsibility for a service is useful for simplifying coordination and spreading load, but it can lead to conflict. It is best suited for distributed systems, like ours, in which limited conflicts are acceptable as long as they are eventually resolved. We now discuss our approach to resolving conflicts arising from distributed probing.

Conflict arises when distributed probing leads to more than one proxy simultaneously managing the same sleeping machine S. This is acceptable, since multiple managers do not interfere in their fundamental duty of waking up S, and reflects our decision to err in favor of ensuring availability rather than consistency. However, it is wasteful of network bandwidth and causes churn in the switches' spanning tree: each manager will reply to ARP requests for S, causing the switch to relearn the port each time. Thus, we take the following steps to eventually detect and resolve multiple management.

We define the *priority* of a proxy P to manage a sleeping machine S as the hash of the concatenation of P and S's MAC addresses. In what follows, assume S is the managee currently managed by M, H is a manager with higher priority, and L is a manager with lower priority.

1. If M hears a message from H indicating it is managing S, M stops managing S and sends an ARP request for S's IP address. The purpose of the ARP request is to trigger H to send an ARP response, thereby taking control of S's port away from M.
2. If M hears a message from L indicating it is managing S, it sends a message to L saying S is managed. This causes L to perform step 1.

We will see in §4.3 that each manager broadcasts a message upon managing a machine, and periodically rebroadcasts it. Therefore, it will eventually trigger one of the conditions above.

Other distributed systems may choose different approaches for conflict resolution, such as exponential backoff. We use priority assignment because during conflict it maintains multiple servicers rather than zero. Specifically, while two or more proxies are deciding which among them should manage a sleeping machine, it remains managed by at least one of them.

4.2.3 Manager duties

When a proxy P manages a sleeping participant S, it acts largely the same as the sleep proxy described in §3.2. For instance, P uses Ethernet port hijacking to monitor traffic to S, maintains S's network presence by responding to ARP requests, and wakes S if it sees a TCP SYN to an open port. However, there are a few notable differences arising from the distributed nature of GreenUp.

First, P monitors not only incoming SYNs destined for S but also *outbound* SYNs destined for S. This is because, unlike the dedicated sleep proxy [23], P is an end-user machine whose user may actually be trying to connect to a service on S. Since P has hijacked S's port, any traffic it sends to S will be dropped by the switch—not sent back to P—to avoid routing loops. Thus, the only way for P to detect a local attempt to connect to S is by monitoring its own outgoing traffic.

Second, as discussed in §4.2.1, P must respond to SYNs for S on the special probing port by saying S is managed. Otherwise, the prober will conclude S is unmanaged and start managing it.

4.3 Subnet state coordination

The distributed probing and management algorithm in §4.2 assumes that each proxy knows the state of each participant in the subnet. Thus, we need participant state to be reliably distributed to all proxies despite our use of unreliable, load-sensitive machines with correlated behavior. In this subsection, we describe our approach to this called *subnet state coordination*. This technique is applicable to any distributed system that needs to disseminate state among a set of machines on the same subnet, but does not require strong consistency for this state.

We chose not to use a fault-tolerant replicated service [6, 14, 25], since a correlated failure such as a power outage or switch failure could subvert its assumption of a fixed fault threshold, rendering it unavailable. Also, the provision of strong consistency is overkill since small differences in proxies' views of the global state do not cause problems. For our system, eventual consistency is sufficient.

Unlike gossip [9, 10], subnet state coordination exploits the fact that all participants are on the same subnet. Thus, (1) they have an efficient broadcast channel, (2) they can wake each other up using the scheme in §3.2, and (3) they are all loosely time-synchronized (*e.g.*, via NTP). Our technique consists of three elements:

1. **Periodic broadcast.** Each participant periodically broadcasts its state, using a period equal to the maximum desirable staleness. It also broadcasts its state whenever it changes.
2. **Rebroadcast by managers.** When a participant is asleep, its manager is responsible for its broadcasts.
3. **Roll call.** Periodically, there is a window during which all asleep participants are awakened. Each proxy drops from its participant list any that do not broadcast during this window.

Periodic broadcast takes advantage of the efficient broadcast channel, and would be sufficient if participants were perfectly reliable. However, since they run on machines that can sleep at any time, rebroadcast by managers is required to ensure that state about sleeping participants is also disseminated. Because a machine that has failed or is no longer participating in GreenUp is hard to distinguish from a sleeping participant, the roll call is used to identify the current set of participants. It ensures that each proxy eventually removes the state of a former participant, and thus stops trying to manage it; as we discuss in our technical report [26], it is acceptable for this to happen eventually rather than immediately. For durability, proxies store their global view of state on disk.

We call the periodic broadcasts of state *heartbeats*. When a participant sends a heartbeat about itself, we call this a *direct heartbeat*. When a participant sends a heartbeat about another participant, we call this an *indirect heartbeat*. Each direct heartbeat includes a timestamp, and each indirect heartbeat includes the timestamp of the last direct heartbeat on which it is based. This way, if a proxy knows more recent state about a participant than the participant's manager, it can inform the manager.

As a side effect of receiving heartbeats, a proxy learns which participants are awake and which are managed. It uses this to compute the sets $\mathcal{S}$ and $\mathcal{K}$ for distributed probing, and also for apocalypse prevention below.

For the roll call, we use a window of ten minutes each day, which is long enough to account for clock skew. Since one of our goals is to let machines sleep as much as possible, we choose a roll call window when most machines are awake anyway, which in our organization is 2:00 pm–2:10 pm.

In environments where there is already a highly-available means for sharing state, like Bigtable [7] or HDFS [27], that system could be used instead of subnet state coordination. However, by using subnet state coordination, our system becomes instantly deployable in any environment with no external dependency.

4.4 Preventing apocalypse

While allowing proxies to sleep increases energy savings, we cannot allow too many to sleep, for two reasons. First, proxies may become overloaded, due to the demands of distributed probing and management. Second, we may reach a state where no proxy is awake, *i.e.*, *apocalypse*. This is a disastrous scenario because users who participate in GreenUp have no way of contacting their machines during apocalypse.

Initially, we thought we could rely on the natural behavior of proxies to prevent apocalypse, and only force them to stay awake when their number reached a dangerously low threshold. However, this approach only works if machines sleep independently, which we show is untrue (§4.4.1). Thus, we develop a technique that maintains a minimum number of awake proxies at all times (§4.4.2). This technique can be used by any distributed system to prevent system failures caused by correlated sleep behavior.

4.4.1 Non-independent sleep behavior

Since GreenUp is targeted for corporate deployment, we require apocalypse to occur with extremely low probability. Such a low probability can only be derived if machines go to sleep independently, so their probabilities multiply. That is, if each awake proxy $i \in \mathcal{K}$ goes to sleep with probability p_i, then we would like to say that the probability all $|\mathcal{K}|$ machines sleep is $\prod_{i \in \mathcal{K}} p_i$, which decreases exponentially.

Unfortunately, we were unable to confirm independent sleep behavior from our data. For this analysis, we used data provided to us by Reich et al. [23] tracing the sleep behavior of 51 distinct machines over a 45-day period ending 1/3/2010. We measured the pairwise independence of every pair of machines A, B with at least 100 samples each, by computing the distribution of: (1) the time to the next sleep event of B from a sleep event of A; and (2) the time to the next sleep event of B from a random time. Given sufficient samples, these distributions should be statistically similar if A and B sleep independently. We measured statistical distance using the two-sample Kolmogorov-Smirnov (K-S) test [13, p. 45–56]. At significance level $\alpha = 0.05$, we found that over 61% of the 523 pairs of machines tested failed. We conclude that we cannot rely on the natural behavior of machines to prevent apocalypse, and instead opt for a design that always keeps some machines awake.

4.4.2 Guardians

Since proxies exhibit unreliable, correlated sleep behavior, our approach to prevent apocalypse is to maintain a minimum number of *guardians* at all times, *i.e.*, proxies that prevent themselves from automatically sleeping. They do so by making a special Windows request indicating the machine should not sleep. Maintaining guardians serves a similar purpose to deploying a static set of proxy servers, but with the advantages of zero deployment cost and resilience to failures of particular machines.

Since even guardians can become unavailable, *e.g.*, due to users hitting power buttons, we require multiple guardians to ensure high availability. The number of guardians we use is described by a function $q(n) = \max\{Q, \frac{n}{B}\}$, where n is the number of participants and Q and B are constants. The purpose of Q is to make unlikely the situation that all guardians simultaneously fail. The purpose of B is to limit the probing and management load of each proxy. We discuss our choice of $Q = 3$ in §7.3 and our choice of $B = 100$ in §7.2.

For the sake of load distribution and to avoid relying on individual machines, proxies dynamically choose the current set of $q(n)$ guardians. They use the same deterministic algorithm for this, and thus independently reach the same decision. Specifically, each proxy computes its position in an ordering of the proxies $\mathcal{K}$, by hashing the MAC address of each proxy with the current date. A proxy becomes a guardian if and only if it is in one of the $q(n)$ highest positions. If a proxy ever notices that there are only $q(n) - c$ proxies, it wakes the c asleep participants with the highest positions. If any of these participants does not wake up within a reasonable period, it wakes the next-highest asleep participant, and so on.

The above approach only fails when all guardians stop so close together in time that they cannot detect this, such as during a power outage. Fortunately, when power is restored, the proxies with Wake-on-Power enabled in their BIOS will come back on and use their durably-stored state to bring GreenUp out of apocalypse. Unfortunately, Windows currently does not support programmatically enabling Wake-on-Power, and machines that do not Wake-on-Power and are unable to Wake-on-LAN from an off state cannot be woken after a power outage. Thus, we plan to programmatically enable Wake-on-Power when it becomes available, and also plan to advise users to enable it manually for the time being.

5 Analysis

This section analyzes certain stability, correctness, and efficiency properties of GreenUp. In doing so, we analytically justify several of GreenUp's design decisions, including its key parameter choices. We guide our anal-

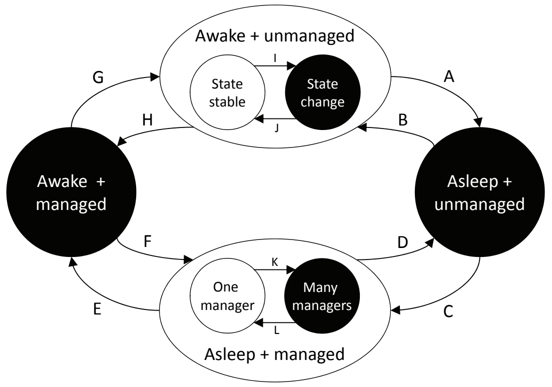


Figure 2: The lifecycle of a machine in GreenUp. Stable states are shown in white.

ysis by modeling the lifecycle of a participant as a state machine with stable and unstable states, shown in Figure 2. Our system maintains the invariant that it converges quickly and reliably to stable states.

5.1 Probe frequency and amount

Let $\mathcal{K}$ be the set of proxies. Every s seconds, each proxy $i \in \mathcal{K}$ sends b_i probes uniformly at random to $n - m_i$ participants, where m_i is the number of participants managed by i, including itself. Declare a *round* of probing to be complete when all $\sum_{i \in \mathcal{K}} b_i$ probes have been sent. We only care about a participant being probed if it is asleep and unmanaged, as otherwise it does not require probing.

Lemma 1. *If $b_i = -(n - m_i)\ln(1-p)/|\mathcal{K}|$ for $0 < p < 1$, then the probability that an asleep, unmanaged participant remains unprobed after a round is at most $1-p$, and the expected number of unprobed participants is concentrated about its mean.*

Proof. Let Z_j be the indicator random variable for participant j remaining unprobed after a round. By assumption, no proxy is j's manager, so all proxies probe j:

$$\begin{aligned}
\Pr(Z_j = 1) &= \prod_{i \in \mathcal{K}} \left(1 - \frac{1}{n - m_i}\right)^{\frac{-(n-m_i)\ln(1-p)}{|\mathcal{K}|}} \\
&\leq \prod_{i \in \mathcal{K}} e^{\frac{\ln(1-p)}{|\mathcal{K}|}} \\
&= \left((1-p)^{\frac{1}{|\mathcal{K}|}}\right)^{|\mathcal{K}|} = 1 - p
\end{aligned}$$

To show concentration about the mean $\mathrm{E}[Z] = \sum_j \mathrm{E}[Z_j] \leq n(1-p)$, we apply Azuma's inequality [17, p. 92] to an appropriate martingale sequence. □

Moreover, the probability a participant is unprobed decreases by $(1-p)$ each round, *i.e.*, exponentially. We set $p = 90\%$ in our implementation, ensuring that transition C in Figure 2 is fast and reliable.

5.2 Management load

We can view participants falling asleep as a balls-in-bins process [21]. Initially, there are n awake proxies and each has a "ball". When proxy i goes to sleep, i and its managees are eventually managed by random proxies due to distributed probing: in effect, i throws m_i balls randomly over the awake proxies. When i wakes up, it stops being managed: in effect, i retrieves its ball.

Lemma 2. *After t sleep events, the distribution of managees on the $n - t$ awake proxies is identical to that of throwing t balls into $n-t$ bins uniformly at random [21].*

Proof. Let $\mathcal{X}_k$ be the set of proxies whose balls are thrown during the k^{th} sleep event, for $1 \leq k \leq t$. We characterize the probability of proxy i's ball landing in proxy j. If $i \in \mathcal{X}_t$, then the probability is $P_t = 1/(n-t)$ since this is the last sleep event. For $1 \leq k \leq t-1$, observing that each proxy's ball is (re)thrown independently of any other ball, the probability is:

$$\begin{aligned}
P_k &= \Pr[i \in \mathcal{X}_k \text{ throws ball in } j] \\
&= \left(\frac{n-t}{n-k}\right)\left(\frac{1}{n-t}\right) + \left(\frac{t-k}{n-k}\right) \sum_{j=k+1}^{t} \frac{P_j}{t-k}
\end{aligned}$$

since either i lands in an awake proxy (one of which is j), or i lands in one of $t-k$ proxies that subsequently sleep. Since $P_t = 1/(n-t)$, this recurrence gives $P_k = 1/(n-t)$ for all $1 \leq k \leq t$.

□

Lemma 2 implies that transitions C and D preserve load balance when a proxy goes to sleep. We now describe two optimizations to these transitions that have not yet been implemented. First, if proxies awaken at random, we can preserve load balance by giving an awakened proxy half of its previous manager's managees. Second, when a proxy is about to sleep, we can avoid the period during which its managees are unmanaged by explicitly handing them off to another proxy chosen at random. The resulting process is closely related to the "coalescent" process in population genetics [12] used to trace the origins of a gene's alleles. We have studied the load properties of the process and have proved the following result, to appear in a forthcoming report:

Theorem 1. *After t random sleep events using the above hand-off policy, the expected maximum number of managees on an awake proxy is less than H_{n-t} times the optimal $n/(n-t)$, where H_i are the harmonic numbers.*

5.3 Other transitions and cycles

Our apocalypse prevention scheme in §4.4.2 ensures that transition C is reliable. The remaining transitions in Figure 2 are easy to argue. Transition J is fast because participants broadcast their state whenever it changes.

The multiple management protocol ensures transition L occurs and prevents transition D by using a deterministic hash on proxy priorities. Cycles (A,B) and (E,F) are unlikely: data from our deployment shows that less than 0.88% of sleep intervals are shorter than 30 seconds (enough time to determine a machine is asleep), and less than 0.68% of awake intervals are less than 10 seconds (plenty of time for a broadcast). Cycles (G,H) and (K,L) are indicative of connectivity issues, which we address in §6. Cycle (A,C,E,G) is rare because views are kept consistent by subnet state coordination and managers only awaken managees for legitimate requests.

6 Implementation

We implemented GreenUp in C#, using four main modules: (1) a state collector, which implements subnet state coordination; (2) a prober, which implements distributed probing; (3) a proxy, which implements traffic interception by interfacing with a packet sniffer; and (4) a logger, which collects debugging information and periodically uploads the logs to a central location. The current implementation of GreenUp is 9,578 lines of code according to SLOCCount [33]. It is packaged with a client UI, not discussed in this paper, that presents information to the user like how often her machine sleeps.

TCP SYNs in probes. Our initial implementation used UDP packets instead of TCP SYNs when probing. However, UDP packets are often encrypted with IPSec using key material shared between the prober and probe target. If the probe target is managed, the manager intercepting its traffic would find these packets undecipherable and would not be able to respond, leading the prober to incorrectly conclude that the target is unmanaged.

To fix this problem, we switched to TCP SYNs, which are not encrypted with IPSec in our network [23]. This solution has an important advantage: When a manager responds to a probe, it demonstrates not only its ability to successfully intercept traffic, but specifically its ability to intercept and recognize TCP SYNs, which is required to wake up a managee on actual connection attempts. This makes our protocol highly robust: If a proxy is not doing a good job of managing a sleeping node, another proxy will detect this and take its place.

Fail-safes. Since GreenUp runs on end-user machines, we implemented some fail-safes for added protection. First, we monitor the moving average of the GreenUp process's CPU utilization and, if it reaches a certain threshold (*e.g.*, 20%), we disable the proxy functionality of that machine temporarily. Second, proxies monitor their suitability for management by making sure they receive regular probes for managees and ensuring connectivity to the subnet's default gateway. If a proxy decides it is unsuitable to manage machines, it stops doing so; as with any type of proxy failure, other proxies will discover the abandoned managees and start managing them.

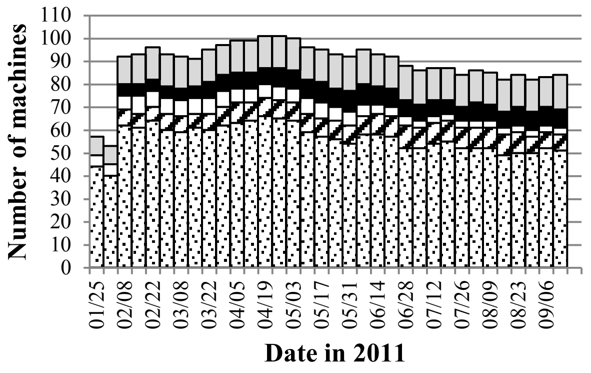


Figure 3: Number of participants each day, broken down by subnet. Each pattern depicts a different subnet.

7 Experiences and Lessons Learned

In this section, we evaluate GreenUp based on our deployment within Microsoft Research. Figure 3 shows the number of users per subnet. At peak, we had 101 participating user machines, with 66 on a single subnet. The jump on February 2 is due to an advertisement e-mail we sent. We see some attrition as people replace their computers and our advertising wanes, but as of September 13 there were still 84 participants. The participants were a mixture of 32-bit and 64-bit Windows 7 machines.

This section has two main goals: to demonstrate that GreenUp is effective and efficient, and to describe lessons we have learned that will help future practitioners. To do so, we answer the following questions. §7.1: Does GreenUp consistently wake machines when their services are accessed? When it does not, why not? §7.2: Is GreenUp scalable to large subnets? §7.3: Does GreenUp succeed at maintaining enough awake machines for the proxy service to be perpetually available? §7.4: How much energy does GreenUp potentially save? §7.5: How well does GreenUp do at waking a machine before the user attempting to connect gives up? For answers to questions unrelated to our distributed approach, such as the frequency of different wakeup causes, see our earlier work [23].

7.1 Machine availability

In this section, we evaluate how effective GreenUp is at meeting its main goal, waking machines when there are attempts to connect to them. First, we describe an availability experiment we conducted on GreenUp machines (§7.1.1). Then, since we find WoL failures to be a common cause of unavailability, we analyze GreenUp history to determine how often they occur (§7.1.2).

7.1.1 Availability experiment

To test the availability of machines running GreenUp, we used the following experiment. For each participating machine, we do the following from a single client: First, the client pings the machine, and records whether any response is received. Next, the client attempts to establish

Experiment start	Already awake	Woken	Unwakeable
8pm Sat, Mar 12	47	31	1
5pm Sun, Mar 13	56	24	1
9pm Mon, Mar 14	56	26	0
9pm Tue, Mar 15	55	26	0
10pm Wed, Sep 21	35	28	0
10pm Thu, Sep 22	44	19	0
6pm Sat, Sep 24	37	27	1
5pm Sun, Sep 25	38	27	0
11pm Mon, Sep 26	42	24	1
9pm Tue, Sep 27	41	25	1
10pm Wed, Sep 28	45	21	0

Table 1: Results of trying to wake GreenUp machines by connecting to them. WoL issues on two machines cause four failures; a temporary network problem causes one.

an SMB connection (TCP, port 139) to the machine. Barring a few exceptions, all machines in Microsoft have this port open. Finally, the client pings the machine again. If the machine responds to the first set of pings, we consider it *already awake*; otherwise, if it responds to the second set, we consider it *woken*; otherwise, we consider it *unwakeable*. In cases where the machine is unwakeable, we investigate whether it was in a state from which its owner would not expect GreenUp to wake it, and in that case do not count it. For instance, we do not count instances in which a machine was shut down, hibernated, removed from the network, or broken.

We exclude from consideration machines that appear to be laptops, since we cannot distinguish unwakeability from lack of physical connectivity. GreenUp is not intended for laptops unless they are tethered like desktops, but some users installed it on mobile laptops anyway.

Results of repeated runs of this experiment are shown in Table 1. We find that in 5 cases, 0.6% of the total, GreenUp failed to wake the machine.

Four failures occurred because two machines, one time each, entered a state in which WoL packets sent by GreenUp did not wake them. In each case, the state lasted long enough for two consecutive wake attempts to fail. For the machine unwakeable on March 12 and 13, the unwakeability seems to have been caused by GreenUp sending it a WoL packet too soon after it had become unavailable: 5 sec and 20 sec, respectively.

Our logs indicate that the fifth failure occurred because of an aberrant network condition that lasted for 36 minutes. During this time, the machine's manager was not receiving TCP SYNs from machines outside the subnet, but was receiving them from machines inside the subnet.

Our logs show that the data from this run is representative of the entire seven month deployment period. Thus, we conclude that GreenUp is extremely reliable. When it does fail to wake up a machine, it is typically because a WoL packet sent by the manager fails to wake up the managee. While these failures are rare, they merit more investigation, which we do next.

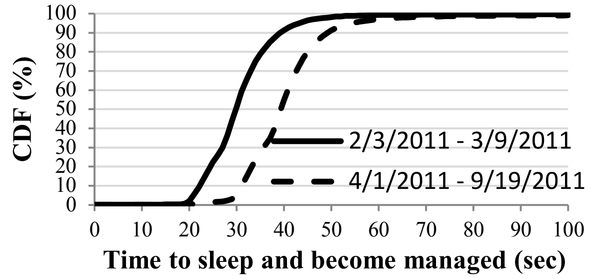


Figure 4: CDF of the time between when a machine begins to sleep and when it becomes managed, before and after deployment of a delay to mitigate WoL bugs.

7.1.2 Wake-on-LAN failures

We now investigate WoL failures in more detail. In many cases, we were able to trace the problem to out-of-date driver, firmware, or NIC hardware. However, we also discovered a rare WoL failure mode affecting a small subset of machines in our deployment. These machines respond correctly to WoL under normal conditions, but if we attempt to wake them up while they are in the process of sleeping, they became permanently unwakeable via WoL until the next reboot! Further tests revealed that the issue can generally be avoided if we wait at least 30 seconds after a machine becomes unresponsive before attempting a wakeup.

Thus, in mid-March we deployed an update that spends 25 seconds probing a machine before deciding to manage it instead of our original timeout of 15 seconds. This has the side benefit of reducing the likelihood that a manager mistakenly manages an awake machine, but it also increases the time to wake a machine in the unlikely event that an access occurs just as the machine falls asleep or its manager becomes unavailable. Fortunately, this event should be rare: Figure 4 shows that a machine is nearly always managed within a minute of beginning to sleep, and a minute is a small fraction of 3.6 hours, the average period a machine spends asleep.

We measured the overall frequency of WoL failures by searching GreenUp logs for events where a machine issues a WoL but the target does not wake up within 180 seconds—enough time for even the slowest-waking machines. However, we must be careful to exclude false positives, such as attempts to wake up a machine that has been shut down. We find that the WoL failure rate is not statistically different before and after the update, mainly because it is tiny to begin with: in the entire period from Feb 3 to Sep 19, 0.3% of WoL attempts are failures. We conclude that WoL is a generally reliable mechanism.

7.2 Scalability

We now evaluate the overhead of running GreenUp, including CPU utilization and network bandwidth, and the implications for scalability to large subnets.

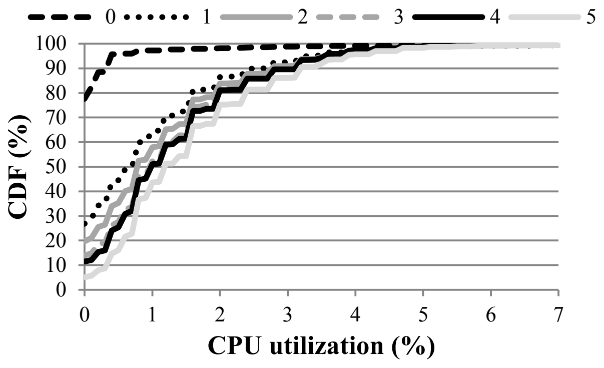


Figure 5: CDF of CPU utilization of GreenUp. Each line represents a different number of managed machines.

# of managees	CPU utilization
100	12%
200	21%
300	29%

Table 2: CPU utilization on a testbed machine as a result of simulating different management loads.

7.2.1 Observed CPU utilization

One concern about GreenUp is the load it places on each participating machine. Qualitatively, we note that our users have never complained about excessive load.

To evaluate the load quantitatively, we had GreenUp record, every ten minutes, the CPU utilization of its process. Figure 5 shows the results in CDF form: for a given number of managees, it shows the CDF of all CPU utilization observations made while the observer was managing that number of managees.

We see that CPU utilization jumps notably when the number of managees goes from zero to one. The median goes from 0.0% to 0.8%, the mean goes from 0.2% to 1.1%, and the 95th percentile utilization goes from 0.4% to 3.5%. This is because most of the CPU overhead is due to parsing sniffed packets, and this is only necessary when managing machines. We see that the overall overhead from parsing these packets is minor, increasing CPU utilization by less than 1%.

We also observe that managing additional machines increases overhead only slightly: going from one managed machine to five managed machines increases mean CPU utilization from 1.1% to 1.3%, and increases 95th percentile utilization from 3.5% to 3.9%. It is difficult to extrapolate from this data since it occupies such a small range, but it suggests that managing 100 machines would yield average CPU utilization of about 13%. In the following subsection, we present another way of estimating the overhead of managing multiple machines.

7.2.2 Scalability of CPU utilization

For this experiment, we simulate the load of managing a large number of machines in a small testbed. We use three machines, each a 32-bit Windows 7 machine with a 2.4-GHz Intel Core 2 6600, 4 GB of memory, and a 1 Gb/s Ethernet card. One goes to sleep, one manages the sleeping machine, and one just sends probes. We increase the probe rate so that the manager sends and receives as many probes as it would if there were n sleeping participants per proxy. We disable the load fail-safe discussed in §6 so we can observe CPU utilization no matter how high it gets.

Note that there are other, minor sources of overhead from managing other machines that we do not simulate. We do not simulate sending heartbeats for each managee, since sending one heartbeat per managee every five minutes is dwarfed by the overhead of sending ~0.8 probes per managee every second. We also do not simulate replying to ARP and ND requests, since we have found such requests to be rare: fewer than 0.05/sec per managee. Finally, we do not simulate parsing SYNs destined for managees since these are even rarer.

Table 2 shows the load on the manager, which we note is somewhat underpowered compared to the typical desktop in our organization. We see that even for this machine, its average CPU utilization when managing 100 machines is 12%, well under our target 20% cap. Recall that we ensure that at least one proxy is always awake for every 100 participants in the subnet, so this shows that load is kept at a reasonable rate. The table also shows that as the number of managees rises to 200 and above, load becomes less reasonable, arguing for our requirement of at least one proxy per 100 participants.

Note that we could reduce this load substantially by increasing the probing period. For instance, changing s from 3 sec to 6 sec would reduce the load by roughly half. The cost is that it would take slightly longer for an unmanaged sleeping machine to become managed. This trade-off hardly seems worthwhile given that we rarely expect a manager to have even close to 100 managees. After all, as we show in §7.3, a far greater fraction of machines than 1/100 tend to be awake at any given time.

7.2.3 Network utilization

Another factor impacting scalability is network utilization. Each GreenUp participant sends and receives probes and heartbeats, and in some cases this traffic scales with the number of participants on the subnet. Qualitatively, this overhead is tiny; we have not received any complaints about GreenUp's network utilization.

To evaluate network utilization analytically, we begin by noting the sizes of various packets we use. The probe SYNs, UDP responses, and ICMP pings are all less than 100 bytes including all headers. The heartbeat packet size depends on the number of ports reported; on our main subnet, the maximum is 554 bytes and the average is 255 bytes. A simple calculation, omitted due to lack of space, shows that even if a machine were to manage

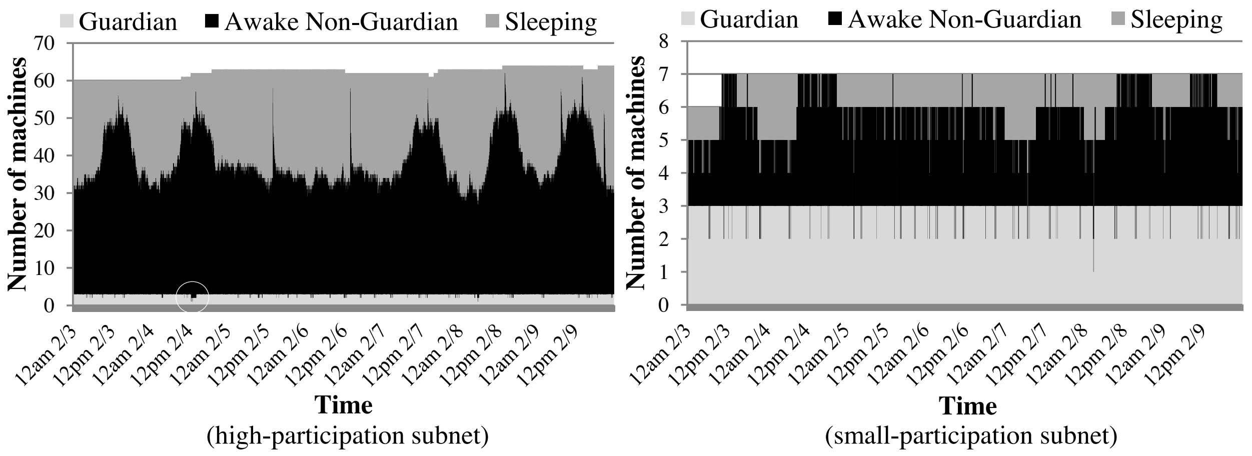


Figure 6: For two different subnets, the number of machines in various roles over time. A circle overlayed on the left graph shows an anomalously long period of time with fewer than three guardians, due to a bug in the implementation.

100 participants in a 1,000-node subnet, it would consume less than 90 Kb/s of upload bandwidth and 60 Kb/s of download bandwidth. Since typical enterprise LANs have 1 Gb/s capacity, this load is negligible.

7.3 GreenUp availability

We now evaluate the effectiveness of GreenUp's guardianship mechanism. Recall that it aims to keep at least three machines awake on each subnet.

The methodology for our evaluation is as follows. Every minute, from a single server, we ping every participating machine. If it does not respond, we consider it asleep. Otherwise, we consider it a guardian if its logs indicate it was a guardian then.

Figure 6 shows the results for the subnet with the largest participation and for a representative subnet with low participation. We see that our guardianship protocol works as expected: generally, there are three guardians, and when occasionally a guardian becomes unavailable, it either returns to availability or is replaced. In one instance, a guardian failed in an undetected manner and was not replaced. This is highlighted in the figure with a white circle: February 4 between 12:46 pm and 2:10 pm. We traced this to a bug in the implementation, which we promptly fixed: if GreenUp is uninstalled from a guardian machine but the machine remains awake, others incorrectly believe it remains a guardian.

Another important element we observe is that at all times a large number of machines are awake even though they are not guardians. This observation suggests that we could save energy by preferentially selecting machines as guardians if they would be awake anyway; this is something we plan to do in future work. By prioritizing already-awake machines to be guardians, we should be able to fully eliminate the energy cost of keeping guardians awake. On the other hand, this observation also shows that choosing random machines to keep awake has a lower cost than expected. Since at least half of non-guardian machines are typically awake, this means that at least half the time a machine is a guardian it would have been awake anyway.

7.4 Energy savings

To evaluate how much energy GreenUp can save, we measure the amount of time GreenUp participants spend asleep. This is readily deduced from logs that indicate when machines went to sleep and woke up. Figure 7 shows the results for Feb 3 through Sep 19; overall, the average participant spends 31% of its time asleep.

An alternative approach to GreenUp's goal of high availability is to not let machines sleep. Thus, by using GreenUp rather than keeping machines awake, we increase sleep time by approximately 31% per day. If the average idle power consumed by desktops is 65 W, then GreenUp saves approximately 175 kWh per machine per year; at 10¢/kWh, this is $17.50 per machine per year.

It would be useful to evaluate whether GreenUp induces users to let their computers sleep more. However, as discussed in §2, the IT department in our organization mandates and enforces a common sleep policy, and users must go to great lengths to change machine sleep behavior. Thus, for our users the benefit of GreenUp is availability, not energy savings. In future work, we would like to evaluate whether, in an organization without such mandates, GreenUp induces users to choose more aggressive sleep policies and thereby save energy.

7.5 User patience

GreenUp's design is predicated on the assumption that users trying to connect to a sleeping machine will continue trying long enough for GreenUp to detect this and wake the machine. To validate this, we now characterize such user patience, and compare it to how long GreenUp takes to wake machines.

To evaluate user patience, we exploit a serendipitous side effect of the occasional unreliability of WoL. When

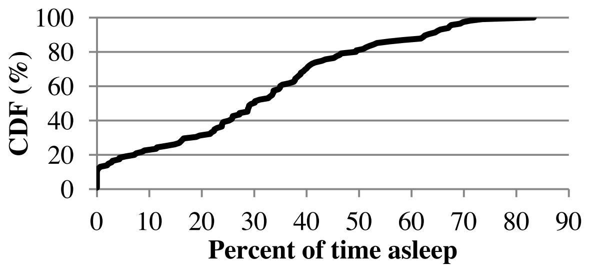


Figure 7: CDF showing distribution, among GreenUp participants, of fraction of time spent asleep.

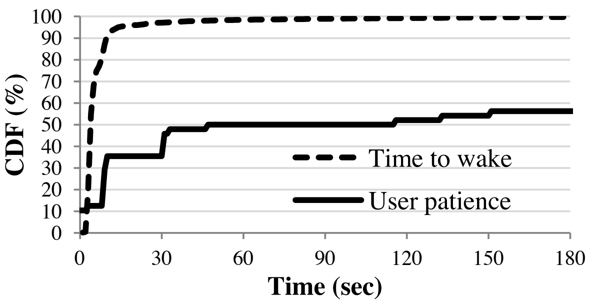


Figure 8: Comparison of how long GreenUp takes to wake machines with how long it has to wake machines, based on evaluation of user patience

a manager detects a TCP connection attempt, but the WoL packet it sends has no effect, the manager's logs show just how long the user continued trying, in vain, to connect to the machine.

In our trace data from Feb 3 through Sep 19, we found 48 events in which there was a connection attempt that allowed us to gauge user patience. Figure 8 shows the results. We find the CDF to be a step function, which is unsurprising given the behavior of TCP: after the first SYN, it sends the next four SYNs roughly 3 sec, 9 sec, 21 sec, and 35 sec later. It appears that 10% of connection attempts send only the first SYN and thus provide GreenUp no time at all to wake the machine. Most likely, many of these are port scans from the IT department's security system, but since we cannot definitively tell this we conservatively estimate them to be highly impatient users. A further 3% send only two SYNs, giving GreenUp 3 sec to wake the machine. A further 22% send only three SYNs, giving GreenUp 9 sec. The remaining 65% send at least five SYNs, giving GreenUp more than 30 sec. Indeed, the most patient 44% seem to try connecting for over five minutes.

Figure 8 also shows how long it takes for GreenUp to wake machines via WoL, using the methodology from §7.1.2. We see that 87% of wakeups take 9 sec or less, so even the 22% of users who are so impatient as to wait only this long can often be satisfied. A full 97% of wakeups take 30 sec or less, so the 65% of users who wait at least this long can nearly always be satisfied. Naturally, the 44% of users who wait at least five minutes are satisfied as long as GreenUp eventually wakes the machine, which as we showed earlier happens over 99% of the time.

Convolving the distribution of user patience with wake time, we find that GreenUp will wake the machine quickly enough to satisfy user patience about 85% of the time. This is about as good as can be expected given that 13% of the time users, or possibly port scanners appearing to be users, are so impatient as to stop trying after only 3 sec. It may be useful, in future deployments, to convey to users the value of waiting longer than this for a connection.

8 Discussion and Future Work

The feedback on GreenUp has been positive, and a larger deployment on ~1,100 machines is currently ongoing. We also continue to improve GreenUp and our users' experience. This section discusses some key issues and areas of ongoing work.

Dynamic layer 2 routing. GreenUp uses spoofed packets to make switches send packets meant for a sleeping machine to its manager. One may view this as meddling with the layer 2 routing protocol, but this is a very common technique, and has been used for many years. Indeed, exactly the same process takes place when a machine is unplugged from one Ethernet port and plugged into another. Some organizations disallow such port changes, but this is not common because it prevents employees from moving machines around without approval from IT staff. In our experience, no switches have exhibited adverse behavior due to such routing changes.

Handling encrypted traffic. GreenUp does not work if all traffic on the network is encrypted. Indeed, we had to implement special exceptions [26] when our organization began using DirectAccess [15], an alternative to VPN that encrypts all traffic with IPSec. We note that this shortcoming is shared by centralized schemes [18, 23] as well as NIC-based schemes [2]. We are currently devising techniques to improve our handling of encrypted traffic. For example, a manager can identify IPSec key exchange traffic to a sleeping machine and determine when a new security association is being set up or renewed; in certain scenarios, it may be appropriate to wake the machine. However, the only foolproof way to deal with encrypted traffic is to use a system such as LiteGreen [8], or to modify the client to send a specially crafted packet before it attempts to establish a connection to a sleeping machine.

Increasing availability. GreenUp achieves over 99% availability; we want to do better. Since most failures are due to WoL problems, we have built a mechanism to automatically test each participant for persistent WoL issues and inform users. However, we want to get to the

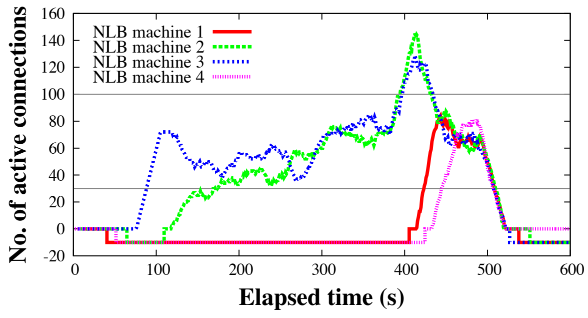


Figure 9: Load profile of a four-machine energy-efficient NLB cluster. A negative load indicates the machine is asleep. Traffic is gradually increased between 75 sec and 475 sec and then stopped.

root of all WoL issues and either directly fix them or inform hardware and software developers how to fix them.

A brief window of unavailability occurs after a sleeping machine's manager becomes unavailable and before a new manager picks it up. We described a strategy in §5.2 that eliminates this time via an explicit hand-off message. Analyzing the load properties of this strategy is our main theoretical pursuit.

Choosing better guardians. Instead of choosing random guardians, it would be more efficient to choose machines that are going to be awake anyway. Thus, we are devising a way to predict when a machine is likely to stay awake and use this information to choose guardians. Since the views of different participants may be stale or divergent, a more complicated protocol is required for allowing a machine to stop being a guardian.

Other applications of GreenUp. Although we have focused on the enterprise desktop setting, the techniques used in GreenUp also apply to certain server cluster settings. In particular, we have used GreenUp to build an energy-efficient version of Windows Network Load Balancing (NLB), a distributed software load balancer used by web sites like `microsoft.com`. NLB runs all machines of a cluster all the time; we used GreenUp to adapt the NLB cluster size to the current load. Specifically, we added a load manager module to GreenUp, using ~300 lines of C# code, to put a machine to sleep or wake another machine if the load is too low or too high, respectively. The load manager interacts with NLB via NLB's management interface—no changes are made to the NLB code. We use a priority-based scheme similar to the one for apocalypse prevention to ensure that only one machine goes to sleep or wakes up at at time. This ensures a minimum number of awake machines, and prevents too many from waking up during high load, to save energy.

We tested our prototype on a four-machine NLB cluster with an IIS web server serving static files. A client program requests a 1MB file with increasing frequency in blocks of 100 seconds for 400 seconds. We use the number of active connections as a measure of load and set the low and high load thresholds to 30 and 100, respectively. Figure 9 shows that our simple scheme achieves dynamic energy efficiency. Initially, there is no load so only machine 3 is awake. At 75 sec, it starts handling requests but quickly detects a steep load slope and wakes up machine 2. The two machines handle the load until 395 sec, when both cross the high threshold and hence wake up machines 1 and 4. The load is quickly redistributed until 475 sec, when the experiment stops and all but one machine soon sleep.

9 Related work

We briefly describe recent work and contrast it with ours. This work falls in two categories: handling sleeping machines, and coordination in distributed systems.

9.1 Sleep and wakeup

We have already described the work most closely related to ours [19, 23]. A somewhat different approach is taken by LiteGreen [8], which requires each user's desktop environment to be in a VM. This VM is live-migrated to a central server when the desktop is idle, and brought back to the user's desktop when the user returns. The main drawback of this system is the need to deploy VM technology on all desktops, which can be quite disruptive. In contrast, our system is easier to deploy and requires no central server.

The SleepServer system [3] uses application stubs to run specially-modified applications on a sleep server while the host machine sleeps. While this allows applications such as BitTorrent to keep sessions alive, it requires modifying code and developing stubs for each application. This is a significant barrier to deployment.

Apple offers a sleep proxy for home users, but it works only with Apple hardware. Adaptiva [1] and Verdiem [30] enable system administrators to wake machines up for management tasks, albeit not seamlessly.

We do not discuss data center power management, as that environment is very different from the enterprise.

9.2 Coordination in distributed systems

One way to coordinate actions in a distributed system is with a replicated state machine [25] that tolerates crash failures [14] or Byzantine failures [6]. These protocols are appropriate when strong consistency of distributed state is required and the set of machines is relatively stable. This is because during conditions such as network partitions, they pause operation until a quorum can be attained. Even systems like Zeno [28] that are designed for higher availability require weak quorums to make progress. In our system, availability is the highest priority, and we can tolerate temporary state inconsistency. The set of awake machines is also highly dynamic. Thus, these approaches are inappropriate.

Other techniques for coordination include distributed hash tables [22, 24, 29, 34] and gossip [9, 10]. These solve the problem of disseminating information among a changing set of machines while also providing high availability. However, they are designed for wide-area networks where no efficient broadcast medium is available. Thus, we propose subnet state coordination as a new alternative when the system runs within a subnet.

Like GreenUp, there are other distributed systems that use randomization to spread load. For example, in RAMCloud [20], masters randomly select backup servers to store their replicas. Also, each RAMCloud server periodically pings one other server at random, reporting any failures to the cluster coordinator. Our distributed probing technique shows how to tune the number of probes to guarantee any desired probability of detecting a failure.

10 Conclusions

This paper presented the design of GreenUp, a decentralized system for providing high availability to sleeping machines. GreenUp operates in a unique setting where broadcast is efficient, machines are highly unreliable, and users must be satisfied with the system to use it. Our design addressed these challenges using new techniques for coordinating subnet state and managing asleep machines. Our analysis of the design showed that it achieves desired availability and efficiency properties. We also implemented GreenUp and deployed it on ~100 users' machines, providing many useful lessons and demonstrating GreenUp's practicality and efficacy. Ultimately, we were able to meet the stringent demands of IT departments for a low-cost, low-administration, and highly-reliable system for keeping sleeping machines available.

Acknowledgments

We thank system administration staff from MSIT and MSR for their invaluable help in making this project possible. We also thank our shepherd, Brian Noble, and the anonymous reviewers for their helpful feedback. Finally, we thank all the users who installed GreenUp, allowing us to collect data and refine our code.

References

[1] Adaptiva. Optimize Config Mgr with Adaptiva Green Planet, Client Health, One Site. http://www.adaptiva.com/, 2009.

[2] Y. Agarwal, S. Hodges, R. Chandra, J. Scott, P. Bahl, and R. Gupta. Somniloquy: Augmenting network interfaces to reduce PC energy usage. In *NSDI*, 2009.

[3] Y. Agarwal, S. Savage, and R. Gupta. SleepServer: A software-only approach for reducing the energy consumption of PCs within enterprise environments. In *USENIX*, 2010.

[4] M. Allman, K. Christensen, B. Nordman, and V. Paxson. Enabling an energy-efficient future Internet through selectively connected end systems. In *Hotnets*, 2007.

[5] AMD. Magic packet technology. Technical report, AMD, 1995.

[6] M. Castro and B. Liskov. Practical Byzantine fault tolerance. In *OSDI*, 1999.

[7] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26, 2008.

[8] T. Das, P. Padala, V. Padmanabhan, R. Ramjee, and K. G. Shin. LiteGreen: Saving energy in networked desktops using virtualization. In *USENIX*, 2010.

[9] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry. Epidemic algorithms for replicated database maintenance. In *PODC*, 1987.

[10] P. Eugster, R. Guerraoui, S. Handurukande, P. Kouznetsov, and A.-M. Kermarrec. Lightweight probabilistic broadcast. *ACM Trans. Comput. Syst.*, 21(4), 2003.

[11] Hewlett-Packard Corp., Intel Corp., Microsoft Corp., Phoenix Technologies Ltd., and Toshiba Corp. Advanced Configuration and Power Interface, revison 4.0, 2009.

[12] J. F. C. Kingman. The coalescent. *Stoch. Process. Appl.*, 13, 1982.

[13] D. E. Knuth. *The Art of Computer Programming.* Vol. 2: *Seminumerical Algorithms*. Addison-Wesley, 1988. 2nd ed.

[14] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2), 1998.

[15] Microsoft Corporation. DirectAccess. http://technet.microsoft.com/en-us/network/dd420463.aspx, 2011.

[16] Microsoft Corporation customer. Private communication, 2010.

[17] R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.

[18] S. Nedevschi, L. Popa, G. Iannaccone, S. Ratnasamy, and D. Wetherall. Reducing network energy consumption via sleeping and rate-adaptation. In *NSDI*, 2008.

[19] S. Nedevschi, J. Chandrashekar, J. Liu, B. Nordman, S. Ratnasamy, and N. Taft. Skilled in the art of being idle: Reducing energy waste in networked systems. In *NSDI*, 2009.

[20] D. Ongaro, S. M. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum. Fast crash recovery in RAMCloud. In *SOSP*, 2011.

[21] M. Raab and A. Steger. "Balls into bins" - A simple and tight analysis. In *RANDOM*, 1998.

[22] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *SIGCOMM*, 2001.

[23] J. Reich, M. Goraczko, A. Kansal, and J. Padhye. Sleepless in Seattle no longer. In *USENIX*, 2010.

[24] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. In *Middleware*, 2001.

[25] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Comput. Surv.*, 22(4), 1990.

[26] S. Sen, J. R. Lorch, R. Hughes, C. Garcia, B. Zill, W. Cordeiro, and J. Padhye. GreenUp: A decentralized system for making sleeping machines available. Technical Report MSR-TR-2012-21, Microsoft Research, 2012.

[27] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The Hadoop distributed file system. In *MSST*, 2010.

[28] A. Singh, P. Fonseca, P. Kuznetsov, R. Rodrigues, and P. Maniatis. Zeno: Eventually consistent Byzantine fault tolerance. In *NSDI*, 2009.

[29] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup protocol for Internet applications. In *SIGCOMM*, 2001.

[30] Verdiem Corporation. http://www.verdiem.com/, 2010.

[31] D. Washburn and O. King. How much money are your idle PCs wasting? Forrester Research Reports, 2008.

[32] C. A. Webber, J. A. Roberson, M. C. McWhinney, R. E. Brown, M. J. Pinckard, and J. F. Busch. After-hours power status of office equipment in the USA. *Energy*, 2006.

[33] D. A. Wheeler. Linux kernel 2.6: It's worth more! http://www.dwheeler.com/essays/linux-kernel-cost.html, 2004.

[34] B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. D. Kubiatowicz. Tapestry: A resilient global-scale overlay for service deployment. *IEEE J. Sel. Areas Commun.*, 22(1), 2004.

Jellyfish: Networking Data Centers Randomly

Ankit Singla[†][*], Chi-Yao Hong[†][*], Lucian Popa[♯], P. Brighten Godfrey[†]
[†] University of Illinois at Urbana–Champaign
[♯] HP Labs

Abstract

Industry experience indicates that the ability to incrementally expand data centers is essential. However, existing high-bandwidth network designs have rigid structure that interferes with incremental expansion. We present Jellyfish, a high-capacity network interconnect which, by adopting a random graph topology, yields itself naturally to incremental expansion. Somewhat surprisingly, Jellyfish is more cost-efficient than a fat-tree, supporting as many as 25% more servers at full capacity using the same equipment at the scale of a few thousand nodes, and this advantage improves with scale. Jellyfish also allows great flexibility in building networks with different degrees of oversubscription. However, Jellyfish's unstructured design brings new challenges in routing, physical layout, and wiring. We describe approaches to resolve these challenges, and our evaluation suggests that Jellyfish could be deployed in today's data centers.

1 Introduction

A well provisioned data center network is critical to ensure that servers do not face bandwidth bottlenecks to utilization; to help isolate services from each other; and to gain more freedom in workload placement, rather than having to tailor placement of workloads to where bandwidth is available [22]. As a result, a significant body of work has tackled the problem of building high capacity network interconnects [6, 17–20, 38, 42, 43].

One crucial problem that these designs encounter is incremental network expansion, *i.e.*, adding servers and network capacity incrementally to the data center. Expansion may be necessitated by growth of the user base, which requires more servers, or by the deployment of more bandwidth-hungry applications. Expansion within a data center is possible through either planned over-provisioning of space and power, or by upgrading old servers to a larger number of more powerful but energy-efficient new servers. Planned expansion is a practical strategy to reduce up-front capital expenditure [28].

Industry experience indicates that incremental expansion is important. Consider the growth of Facebook's data center server population from roughly 30,000 in Nov. 2009 to >60,000 by June 2010 [34]. While Facebook has added entirely new data center facilities, much of this growth involves incrementally expanding existing facilities by "adding capacity on a daily basis" [33]. For instance, Facebook announced that it would double the size of its facility at Prineville, Oregon by early 2012 [16]. A 2011 survey [15] of 300 enterprises that run data centers of a variety of sizes found that 84% of firms would probably or definitely expand their data centers in 2012. Several industry products advertise incremental expandability of the server pool, including SGI's Ice-Cube (marketed as "The Expandable Modular Data Center" [5]; expands 4 racks at a time) and HP's EcoPod [24] (a "pay-as-you-grow" enabling technology [23]).

Do current high-bandwidth data center network proposals allow incremental growth? Consider the fat-tree interconnect, as proposed in [6], as an illustrative example. The entire structure is completely determined by the port-count of the switches available. This is limiting in at least two ways. First, it makes the design space very coarse: full bisection bandwidth fat-trees can only be built at sizes 3456, 8192, 27648, and 65536 corresponding to the commonly available port counts of 24, 32, 48, and 64[1]. Second, even if (for example) 50-port switches were available, the smallest "incremental" upgrade from the 48-port switch fat-tree would add 3,602 servers and would require replacing every switch.

There are, of course, some workarounds. One can replace a switch with one of larger port count or oversubscribe certain switches, but this makes capacity distribution constrained and uneven across the servers. One could leave free ports for future network connections [14, 20] but this wastes investment until actual expansion. Thus, without compromises on bandwidth or cost, such topologies are not amenable to incremental growth.

Since it seems that *structure* hinders incremental expansion, we propose the opposite: a random network in-

[*]A coin toss decided the order of the first two authors.

[1]Other topologies have similar problems: a hypercube [7] allows only power-of-2 sizes, a de Bruijn-like construction [39] allows only power-of-3 sizes, etc.

terconnect. The proposed interconnect, which we call **Jellyfish**, is a *degree-bounded*[2] *random graph* topology among top-of-rack (ToR) switches. The inherently sloppy nature of this design has the potential to be significantly more flexible than past designs. Additional components—racks of servers or switches to improve capacity—can be incorporated with a few random edge swaps. The design naturally supports heterogeneity, allowing the addition of newer network elements with higher port-counts as they become available, unlike past proposals which depend on certain regular port-counts [6, 18–20, 38, 42]. Jellyfish also allows construction of arbitrary-size networks, unlike topologies discussed above which limit the network to very coarse design points dictated by their structure.

Somewhat surprisingly, Jellyfish supports *more* servers than a fat-tree [6] built using the same network equipment while providing at least as high per-server bandwidth, measured either via bisection bandwidth or in throughput under a random-permutation traffic pattern. In addition, Jellyfish has lower mean path length, and is resilient to failures and miswirings.

But a data center network that lacks regular structure is a somewhat radical departure from traditional designs, and this presents several important challenges that must be addressed for Jellyfish to be viable. Among these are routing (schemes depending on a structured topology are not applicable), physical construction, and cabling layout. We describe simple approaches to these problems which suggest that Jellyfish could be effectively deployed in today's data centers.

Our key contributions and conclusions are as follows:

- We propose Jellyfish, an incrementally-expandable, high-bandwidth data center interconnect based on a random graph.

- We show that Jellyfish provides quantitatively easier incremental expansion than prior work on incremental expansion in Clos networks [14], growing incrementally at only 40% of the expense of [14].

- We conduct a comparative study of the bandwidth of several proposed data center network topologies. Jellyfish can support 25% more servers than a fat-tree while using the same switch equipment and providing at least as high bandwidth. This advantage increases with network size and switch port-count. Moreover, we propose *degree-diameter optimal graphs* [12] as benchmark topologies for high capacity at low cost, and show that Jellyfish remains within 10% of these carefully-optimized networks.

- Despite its lack of regular structure, packet-level simulations show that Jellyfish's bandwidth can be effectively utilized via existing forwarding technologies that provide high path diversity.

- We discuss effective techniques to realize physical layout and cabling of Jellyfish. Jellyfish may have higher cabling cost than other topologies, since its cables can be longer; but when we restrict Jellyfish to use cables of length similar to the fat-tree, it still improves on the fat-tree's throughput.

Outline: Next, we discuss related work (§2), followed by a description of the Jellyfish topology (§3), and an evaluation of the topology's properties, unhindered by routing and congestion control (§4). We then evaluate the topology's performance with routing and congestion control mechanisms (§5). We discuss effective cabling schemes and physical construction of Jellyfish in various deployment scenarios (§6), and conclude (§7).

2 Related Work

Several recent proposals for high-capacity networks exploit special structure for topology and routing. These include folded-Clos (or fat-tree) designs [6, 18, 38], several designs that use servers for forwarding [19, 20, 45], and designs using optical networking technology [17, 43]. High performance computing literature has also studied carefully-structured expander graphs [27].

However, none of these architectures address the *incremental expansion* problem. For some (including the fat-tree), adding servers while preserving the structural properties would require replacing a large number of network elements and extensive rewiring. MDCube [45] allows expansion at a very coarse rate (several thousand servers). DCell and BCube [19, 20] allow expansion to an *a priori* known target size, but require servers with free ports reserved for planned future expansion.

Two recent proposals, Scafida [21] (based on scale-free graphs) and Small-World Datacenters (SWDC) [41], are similar to Jellyfish in that they employ randomness, but are significantly different than our design because they require correlation (i.e., structure) among edges. This structured design makes it unclear whether the topology retains its characteristics upon incremental expansion; neither proposal investigates this issue. Further, in SWDC, the use of a regular lattice underlying the topology creates familiar problems with incremental expansion.[3] Jellyfish also has a capacity advantage over

[2]Degree-bounded means that the number of connections per node is limited, in this case by switch port-counts.

[3]For instance, using a 2D torus as the lattice implies that maintaining the network structure when expanding an n node network requires adding $\Theta(\sqrt{n})$ new nodes. The higher the dimensionality of the lattice, the more complicated expansion becomes.

both proposals: Scafida has marginally worse bisection bandwidth and diameter than a fat-tree, while Jellyfish improves on fat-trees on both metrics. We show in §4.1 that Jellyfish has higher bandwidth than SWDC topologies built using the same equipment.

LEGUP [14] attacks the expansion problem by trying to find optimal upgrades for Clos networks. However, such an approach is fundamentally limited by having to start from a rigid structure, and adhering to it during the upgrade process. Unless free ports are preserved for such expansion (which is part of LEGUP's approach), this can cause significant overhauls of the topology even when adding just a few new servers. In this paper, we show that Jellyfish provides a simple method to expand the network to almost any desirable scale. Further, our comparison with LEGUP (§4.2) over a sequence of network expansions illustrates that Jellyfish provides significant cost-efficiency gains in incremental expansion.

REWIRE [13] is a heuristic optimization method to find high capacity topologies with a given cost budget, taking into account length-varying cable cost. While [13] compares with random graphs, the results are inconclusive.[4] Due to the recency of [13], we have left a direct quantitative comparison to future work.

Random graphs have been examined in the context of communication networks [31] previously. Our contribution lies in applying random graphs to allow incremental expansion and in quantifying the efficiency gains such graphs bring over traditional data center topologies.

3 Jellyfish Topology

Construction: The Jellyfish approach is to construct a random graph at the top-of-rack (ToR) switch layer. Each ToR switch i has some number k_i of ports, of which it uses r_i to connect to other ToR switches, and uses the remaining $k_i - r_i$ ports for servers. In the simplest case, which we consider by default throughout this paper, every switch has the same number of ports and servers: $\forall i$, $k = k_i$ and $r = r_i$. With N racks, the network supports $N(k-r)$ servers. In this case, the network is a *random regular graph*, which we denote as RRG(N, k, r). This is a well known construct in graph theory and has several desirable properties as we shall discuss later.

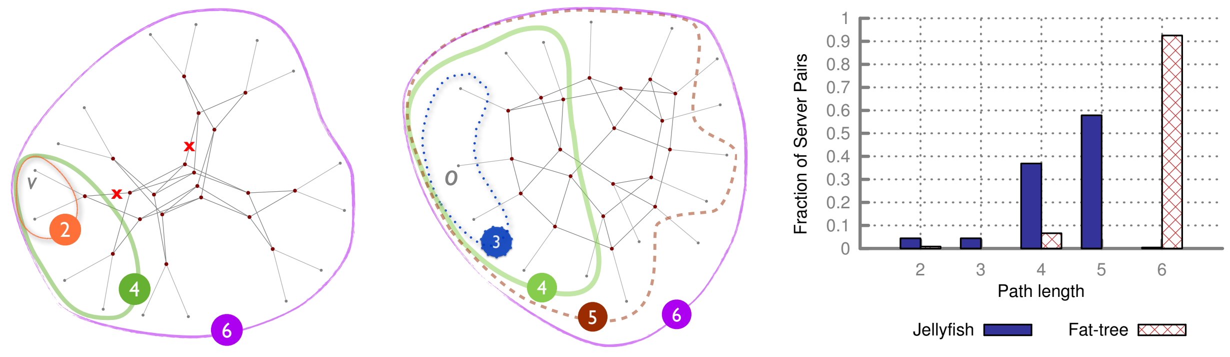
Formally, RRGs are sampled uniformly from the space of all r-regular graphs. This is a complex problem in graph theory [29]; however, a simple procedure produces "sufficiently uniform" random graphs which empirically have the desired performance properties. One can simply pick a random pair of switches with free ports (for the switch-pairs are not already neighbors), join them with a link, and repeat until no further links can be added. If a switch remains with ≥ 2 free ports (p_1, p_2) — which includes the case of incremental expansion by adding a new switch — these can be incorporated by removing a uniform-random existing link (x,y), and adding links (p_1,x) and (p_2,y). Thus only a single unmatched port might remain across the whole network.

Using the above idea, we generate a blueprint for the physical interconnection. (Allowing human operators to "wire at will" may result in poor topologies due to human biases – for instance, favoring shorter cables over longer ones.) We discuss cabling later in §6.

Intuition: Our two key goals are high bandwidth and flexibility. The intuition for the latter property is simple: lacking structure, the RRG's network capacity becomes "fluid", easily wiring up any number of switches, heterogeneous degree distributions, and newly added switches with a few random link swaps.

But why should random graphs have high bandwidth? We show quantitative results later, but here we present the intuition. The end-to-end throughput of a topology depends not only on the capacity of the network, but is also inversely proportional to the *amount of network capacity consumed to deliver each byte* — that is, the average path length. Therefore, assuming that the routing protocol is able to utilize the network's full capacity, low average path length allows us to support more flows at high throughput. To see why Jellyfish has low path length, Fig. 1(a) and 1(b) visualize a fat-tree and a representative Jellyfish topology, respectively, with *identical* equipment. Both topologies have diameter 6, meaning that any server can reach all other servers in 6 hops. However, in the fat-tree, each server can only reach 3 others in ≤ 5 hops. In contrast, in the random graph, the typical origin server labeled o can reach 12 servers in ≤ 5 hops, and 6 servers in ≤ 4 hops. The reason for this is that many edges in the fat-tree are not useful from the perspective of their effect on path length; for example, deleting the two edges marked X in Fig. 1(a) does not increase the path length between any pair of servers. In contrast, the RRG's diverse random connections lead

[4]REWIRE attempts to improve a given "seed" graph. The seed could be a random graph, so in principle [13] should be able to obtain results at least as good as Jellyfish. In [13] the seed was an empty graph. The results show, in some cases, fat-trees obtaining more than an order of magnitude worse bisection bandwidth than random graphs, which in turn are more than an order of magnitude worse than REWIRE topologies, all at equal cost. In other cases, [13] shows random graphs that are disconnected. These significant discrepancies could arise from (a) separating network port costs from cable costs rather than optimizing over the total budget, causing the random graph to pay for more ports than it can afford cables to connect; (b) assuming linear physical placement of all racks, so cable costs for distant servers scale as $\Theta(n)$ rather than $\Theta(\sqrt{n})$ in a more typical two-dimensional layout; and (c) evaluating very low bisection bandwidths (**0.04** to 0.37) — in fact, at the highest bisection bandwidth evaluated, [13] indicates the random graph has higher throughput than REWIRE. The authors indicated to us that REWIRE has difficulty scaling beyond a few hundred nodes.



(a) (b) (c)

Figure 1: *Random graphs have high throughput because they have low average path length, and therefore do less work to deliver each packet.* ***(a):*** *Fat-tree with* 16 *servers and* 20 *four-port switches.* ***(b):*** *Jellyfish with identical equipment. The servers are leaf nodes; switches are interior nodes. Each 'concentric' ring contains servers reachable from any server v in the fat-tree, and an arbitrary server o in Jellyfish, within the number of hops in the marked label. Jellyfish can reach many more servers in few hops because in the fat tree, many edges (like those marked "X") are redundant from a path-length perspective.* ***(c):*** *Path length distribution between servers for a* 686*-server Jellyfish (drawn from* 10 *trials) and same-equipment fat-tree.*

to lower mean path length.[5] Figure 1(c) demonstrates these effects at larger scale. With 686 servers, >99.5% of source-destination pairs in Jellyfish can be reached in fewer than 6 hops, while the corresponding number is only 7.5% in the fat-tree.

4 Jellyfish Topology Properties

This section evaluates the efficiency, flexibility and resilience of Jellyfish and other topologies. Our goal is to measure the raw capabilities of the topologies, were they to be coupled with optimal routing and congestion control. We study how to perform routing and congestion control separately, in §5.

Our key findings from these experiments are:

- Jellyfish can support 27% more servers at full capacity than a (same-switching-equipment) fat-tree at a scale of <900 servers. The trend is for this advantage to *improve* with scale.
- Jellyfish's network capacity is >91% of the best-known degree-diameter graphs [12], which we propose as benchmark bandwidth-efficient graphs.
- Paths are shorter on average in Jellyfish than in a fat-tree, and the *maximum* shortest path length (diameter) is the same or lower for all scales we tested.
- Incremental expansion of Jellyfish produces topologies identical in throughput and path length Jellyfish topologies generated from scratch.
- Jellyfish provides a significant cost-efficiency advantage over prior work (LEGUP [14]) on incremental network expansion in Clos networks. In a network expansion scenario that was made available for us to test, Jellyfish builds a slightly higher-capacity expanded network at only 40% of LEGUP's expense.
- Jellyfish is highly failure resilient, even more so than the fat-tree. Failing a random 15% of all links results in a capacity decrease of <16%.

Evaluation methodology: Some of the results for network capacity in this section are based on explicit calculations of the theoretical bounds for bisection bandwidth for regular random graphs.

All throughput results presented in this section are based on calculations of throughput for a specific class of traffic demand matrices with optimal routing. The traffic matrices we use are *random permutation traffic*: each server sends at its full output link rate to a single other server, and receives from a single other server, and this permutation is chosen uniform-randomly. Intuitively, random permutation traffic represents the case of no locality in traffic, as might arise if VMs are placed without regard to what is convenient for the network[6]. Nevertheless, evaluating other traffic patterns is an important question that we leave for future work.

Given a traffic matrix, we characterize a topology's raw capacity with "ideal" load balancing by treating flows as splittable and fluid. This corresponds to solving a standard multi-commodity network flow problem. (We use the CPLEX linear program solver [1].)

For all throughput comparisons, we use the *same switching equipment* (in terms of both number of switches, and ports on each switch) for each set of

[5] This is related to the fact that RRGs are *expander graphs* [11].

[6] Supporting such flexible network-oblivious VM placement without a performance penalty is highly desirable [22].

topologies compared. Throughput results are always normalized to $[0,1]$, and averaged over all flows.

For comparisons with the full bisection bandwidth fat-tree, we attempt to find, using a binary search procedure, the maximum number of servers Jellyfish can support using the same switching equipment as the fat-tree while satisfying the full traffic demands. Specifically, each step of the binary search checks a certain number of servers m by sampling three random permutation traffic matrices, and checking whether Jellyfish supports full capacity for *all* flows in *all* three matrices. If so, we say that Jellyfish supports m servers at full capacity. After our binary search terminates, we verify that the returned number of servers is able to get full capacity over each of 10 more samples of random permutation traffic matrices.

4.1 Efficiency

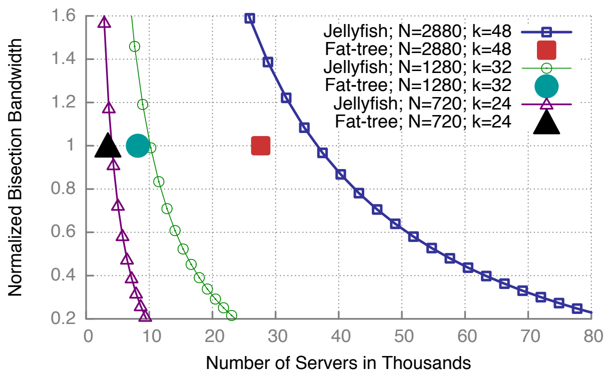
Bisection Bandwidth vs. Fat-Tree: Bisection bandwidth, a common measure of network capacity, is the *worst-case* bandwidth spanning any two equal-size partitions of a network. Here, we compute the fat-tree's bisection bandwidth directly from its parameters; for Jellyfish, we model the network as a RRG and apply a lower bound of Bollobás [8]. We normalize bisection bandwidth by dividing it by the total line-rate bandwidth of the servers in one partition[7].

Fig. 2(a) shows that at the same cost, Jellyfish supports a larger number of servers (x axis) at full bisection bandwidth (y axis $= 1$). For instance, at the same cost as a fat-tree with 16,000 servers, Jellyfish can support $>$20,000 servers at full bisection bandwidth. Also, Jellyfish allows the freedom to accept lower bisection bandwidth in exchange for supporting more servers or cutting costs by using fewer switches.

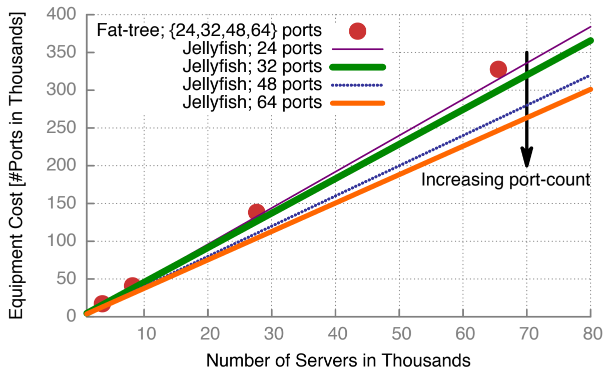
Fig. 2(b) shows that the cost of building a full bisection-bandwidth network increases more slowly with the number of servers for Jellyfish than for the fat-tree, especially for high port-counts. Also, the design choices for Jellyfish are essentially continuous, while the fat-tree (following the design of [6]) allows only certain discrete jumps in size which are further restricted by the port-counts of available switches. (Note that this observation would hold even for over-subscribed fat-trees.)

Jellyfish's advantage increases with port-count, approaching twice the fat-tree's bisection bandwidth. To see this, note that the fat-tree built using k-port switches has $k^3/4$ servers, and being a full-bisection interconnect, it has $k^3/8$ edges crossing each bisection. The fat-tree has $k^3/2$ switch-switch links, implying that its bisection bandwidth represents $\frac{1}{4}$ of its switch-switch links. For Jellyfish, in expectation, $\frac{1}{2}$ of its switch-switch links

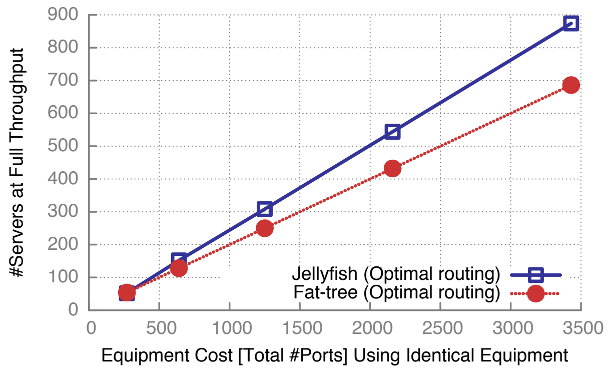
[7]Values larger than 1 indicate overprovisioning.



(a)



(b)



(c)

Figure 2: *Jellyfish offers virtually continuous design space, and packs in more servers at high network capacity at the same expense as a fat-tree. From theoretical bounds: (a) Normalized bisection bandwidth versus the number of servers supported; equal-cost curves, and (b) Equipment cost versus the number of servers for commodity-switch port-counts* (24, 32, 48) *at full bisection bandwidth. Under optimal routing, with random-permutation traffic: (c) Servers supported at full capacity with the same switching equipment, for* 6, 8, 10, 12 *and* 14*-port switches. Results for (c) are averaged over* 8 *runs.*

cross any given bisection of the switches, which is *twice* that of the fat-tree assuming they are built with the same number of switches and servers. Intuitively, Jellyfish's worst-case bisection should be slightly worse than this average bisection. The bound of [8] bears this out: in almost every r-regular graph with N nodes, every set of $N/2$ nodes is joined by at least $N(\frac{r}{4} - \frac{\sqrt{r \ln 2}}{2})$ edges to the rest of the graph. As the number of network ports $r \to \infty$ this quantity approaches $Nr/4$, i.e., $\frac{1}{2}$ of the $Nr/2$ links.

Throughput vs. Fat Tree: Fig. 2(c) uses the random-permutation traffic model to find the number of servers Jellyfish can support at full capacity, matching the fat-tree in capacity and switching equipment. The improvement is as much as 27% more servers than the fat-tree at the largest size (874 servers) we can use CPLEX to evaluate. As with results from Bollobás' theoretical lower bounds on bisection bandwidth (Fig. 2(a), 2(b)), the trend indicates that this improvement increases with scale.

Throughput vs. Degree-Diameter Graphs: We compare Jellyfish's capacity with that of the best known degree-diameter graphs. Below, we briefly explain what these graphs are, and why this comparison makes sense.

There is a fundamental trade-off between the degree and diameter of a graph of a fixed vertex-set (say of size N). At one extreme is a clique — maximum possible degree ($N-1$), and minimum possible diameter (1). At the other extreme is a disconnected graph with degree 0 and diameter ∞. The problem of constructing a graph with maximum possible number N of nodes while preserving given diameter and degree bounds is known as the *degree-diameter problem* and has received significant attention in graph theory. The problem is quite difficult and the optimal graphs are only known for very small sizes: the largest degree-diameter graph known to be optimal has $N = 50$ nodes, with degree 7 and diameter 2 [12]. A collection of optimal and best known graphs for other degree-diameter combinations is maintained at [12].

The degree-diameter problem relates to our objective in that short average path lengths imply low resource usage and thus high network capacity. Intuitively, the best known degree-diameter topologies should support a large number of servers with high network bandwidth and low cost (small degree). While we note the distinction between average path length (which relates more closely to the network capacity) and diameter, degree-diameter graphs will have small average path lengths too.

Thus, we propose the best-known degree-diameter graphs as a benchmark for comparison. Note that such graphs do not meet our incremental expansion objectives; we merely use them as a capacity benchmark for Jellyfish topologies. But these graphs (and our measurements of them) may be of independent interest since they could be deployed as highly efficient topologies in a setting where incremental upgrades are unnecessary, such as a pre-fab container-based data center.

For our comparisons with the best-known degree-diameter graphs, the number of servers we attach to the switches was decided such that full-bisection bandwidth

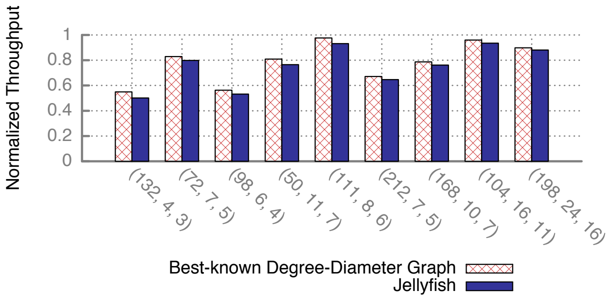


Figure 3: *Jellyfish's network capacity is close to (i.e., ~91% or more in each case) that of the best-known degree-diameter graphs. The x-axis label (A, B, C) represents the number of switches (A), the switch port-count (B), and the network degree (C). Throughput is normalized against the non-blocking throughput. Results are averaged over* 10 *runs.*

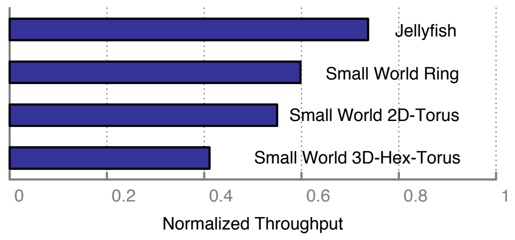


Figure 4: *Jellyfish has higher capacity than the (same-equipment) small world data center topologies [41] built using a ring, a* 2*D-Torus, and a* 3*D-Hex-Torus as the underlying lattice. Results are averaged over* 10 *runs.*

was not hit for the degree-diameter graphs (thus ensuring that we are measuring the full capacity of degree-diameter graphs.) Our results, in Fig. 3, show that the best-known degree-diameter graphs do achieve higher throughput than Jellyfish, and thus improve even more over fat-trees. But in the worst of these comparisons, Jellyfish still achieves ~91% of the degree-diameter graph's throughput. While graphs that are optimal for the degree-diameter problem are not (known to be) provably optimal for our bandwidth optimization problem, these results strongly suggest that Jellyfish's random topology leaves little room for improvement, even with very carefully-optimized topologies. And what improvement is possible may not be worth the loss of Jellyfish's incremental expandability.

Throughput vs. small world data centers (SWDC): SWDC [41] proposes a new topology for data centers inspired by a small-world distribution. We compare Jellyfish with SWDC using the same degree-6 topologies described in the SWDC paper. We emulate their 6-interface server-based design by using switches connected with 1 server and 6 network ports each. We build the three SWDC variants described in [41] at topology sizes as close to each other as possible (constrained by the lattice structure underlying these topologies) across sizes

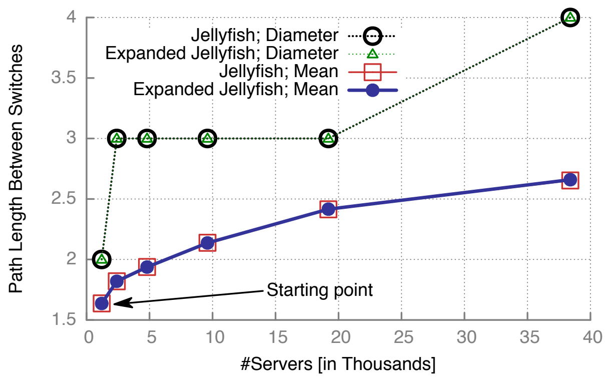


Figure 5: *Jellyfish has short paths: Path length versus number of servers, with* $k = 48$ *port switches of which* $r = 36$ *connect to other switches and* 12 *connect to servers. Each data point is derived from* 10 *graphs. The diameter is* ≤ 4 *at these scales. This figure also shows that constructing Jellyfish from scratch, or using incremental growth yields topologies with very similar path length characteristics.*

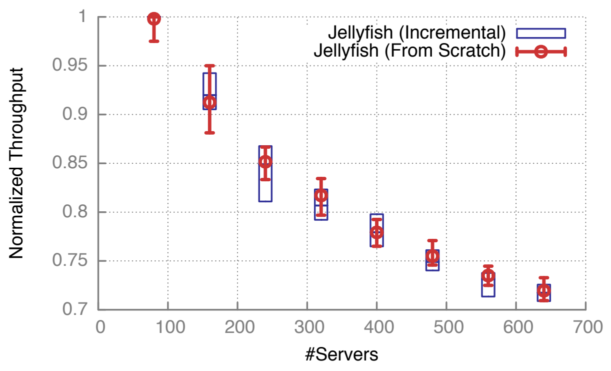


Figure 6: *Incrementally constructed Jellyfish has the same capacity as Jellyfish built from scratch: We built a Jellyfish topology incrementally from* 20 *to* 160 *switches in increments of* 20 *switches, and compared the throughput per server of these incrementally grown topologies to Jellyfish topologies built from scratch using our construction routine. The plot shows the average, minimum and maximum throughput over* 20 *runs.*

we can simulate. Thus, we use 484 switches for Jellyfish, the SWDC-Ring topology, and the SWDC-2D-Torus topology; for the SWDC-3D-Hex-Torus, we use 450 nodes. (Note that this gives the latter topology an advantage, because it uses the same degree, but a smaller number of nodes. However, this is the closest size where that topology is well-formed.) At these sizes, the first three topologies all yielded full throughput, so, to distinguish between their capacities, we oversubscribed each topology by connecting 2 servers at each switch instead of just one. The results are shown in Fig. 4. Jellyfish's throughput is $\sim$119% of that of the closest competitor, the ring-based small world topology.

Path Length: Short path lengths are important to ensure low latency, and to minimize network utilization. In this context, we note that the theoretical *upper-bound* on the diameter of random regular graphs is fairly small: Bollobás and de la Vega [10] showed that in almost every r-regular graph with N nodes, the diameter is at most $1 + \lceil \log_{r-1}((2+\varepsilon)rN \log N)\rceil$ for any $\varepsilon > 0$. Thus, the server-to-server diameter is at most $3 + \lceil \log_{r-1}((2+\varepsilon)rN \log N)\rceil$. Thus, the path length increases logarithmically (base r) with the number of nodes in the network. Given the availability of commodity servers with large port counts, this rate of increase is very small in practice.

We measured path lengths using an all-pairs shortest-paths algorithm. The average path length (Fig. 5) in Jellyfish is much smaller than in the fat-tree[8]. For example, for RRG(3200, 48, 36) with 38,400 servers, the average path length between switches is <2.7 (Fig. 5), while the fat-tree's average is 3.71 at the smallest size, 3.96 at the size of 27,648 servers. Even though Jellyfish's diameter is 4 at the largest scale, the 99.99th percentile path-length across 10 runs did not exceed 3 for any size in Fig. 5.

[8]Note that the results in Fig. 5 use 48-port switches throughout, meaning that the only point of direct, fair comparison with a fat-tree is at the largest scale, where Jellyfish still compares favorably against a fat-tree built using 48-port switches and 27,648 servers.

4.2 Flexibility

Arbitrary-sized Networks: Several existing proposals admit only the construction of interconnects with very coarse parameters. For instance, a 3-level fat-tree allows only $k^3/4$ servers with k being restricted to the port-count of available switches, unless some ports are left unused. This is an arbitrary constraint, extraneous to operational requirements. In contrast, Jellyfish permits any number of racks to be networked efficiently.

Incremental Expandability: Jellyfish's construction makes it amenable to incremental expansion by adding either servers and/or network capacity (if not full-bisection bandwidth already), with increments as small as one rack or one switch. Jellyfish can be expanded such that rewiring is limited to the number of ports being added to the network; and the desirable properties are maintained: high bandwidth and short paths at low cost.

As an example, consider an expansion from an RRG(N, k, r) topology to RRG($N+1$, k, r). In other words, we are adding one rack of servers, with its ToR switch u, to the existing network. We pick a random link (v,w) such that this new ToR switch is not already connected with either v or w, remove it, and add the two links (u,v) and (u,w), thus using 2 ports on u. This process is repeated until all ports are filled (or a single odd port remains, which could be matched with another free port on an existing rack, used for a server, or left free). This completes incorporation of the rack, and can be repeated for as many new racks as desired.

A similar procedure can be used to expand network capacity for an under-provisioned Jellyfish network. In this case, instead of adding a rack with servers, we only add the switch, connecting all its ports to the network.

Jellyfish also allows for heterogeneous expansion: nothing in the procedure above requires that the new switches have the same number of ports as the existing switches. Thus, as new switches with higher port-counts become available, they can be readily used, either in racks or to augment the interconnect's bandwidth. There is, of course, the possibility of taking into account heterogeneity explicitly in the random graph construction and to improve upon what the vanilla random graph model yields. This endeavor remains future work for now.

We note that our expansion procedures (like our construction procedure) may not produce uniform-random RRGs. However, we demonstrate that the path length and capacity measurements of topologies we build incrementally match closely with ones constructed from scratch. Fig. 5 shows this comparison for the average path length and diameter where we start with an RRG with 1,200 servers and expand it incrementally. Fig. 6 compares the normalized throughput per server under a random permutation traffic model for topologies built incrementally against those built from scratch. The incremental topologies here are built by adding successive increments of 20 switches, and 80 servers to an initial topology also with 20 switches and 80 servers. (Throughout this experiment, each switch has 12 ports, 4 of which are attached to servers.) In each case, the results are close to identical.

Network capacity under expansion: Note that after normalizing by the number of servers $N(k-r)$, the lower bound on Jellyfish's normalized bisection bandwidth (§4.1) is independent of network size N. Of course, as N increases with fixed network degree r, average path length increases, and therefore, the demand for additional per-server capacity increases[9]. But since path length increases very slowly (as discussed above), bandwidth per server remains high even for relatively large factors of growth. Thus, operators can keep the servers-per-switch ratio constant even under large expansion, with minor bandwidth loss. Adding only switches (without servers) is another avenue for expansion which can preserve or even increase network capacity. Our below comparison with LEGUP uses both forms of expansion.

Comparison with LEGUP [14]: While a LEGUP implementation is not publicly available, the authors were kind enough to supply a series of topologies produced by LEGUP. In this expansion arc, there is a budget constraint for the initial network, and for each successive expansion step; within the constraint, LEGUP attempts to maximize network bandwidth, and also may keep some ports free in order to ease expansion in future steps. The initial network is built with 480 servers and 34 switches; the first expansion adds 240 more servers and some switches; and each remaining expansion adds only switches. To build a comparable Jellyfish network, at each expansion step, under the same budget constraints, (using the same cost model for switches, *cabling, and rewiring*) we buy and randomly cable in as many new switches as we can. The number of servers supported is the same as LEGUP at each stage.

[9]This discussion also serves as a reminder that bisection-bandwidth, while a good metric of network capacity, is not the same as, say, capacity under worst-case traffic patterns.

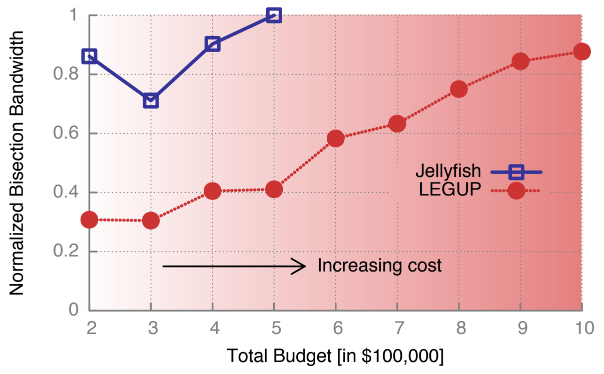


Figure 7: *Jellyfish's incremental expansion is substantially more cost-effective than LEGUP's Clos network expansion. With the same budget for equipment and rewiring at each expansion stage (x-axis), Jellyfish obtains significantly higher bisection bandwidth (y-axis). Results are averaged over* 10 *runs. (The drop in Jellyfish's bisection bandwidth from stage* 0 *to* 1 *occurs because the number of servers increases in that step.)*

LEGUP optimizes for bisection bandwidth, so we compare both LEGUP and Jellyfish on that metric (using code provided by the LEGUP authors [14]) rather than on our previous random permutation throughput metric. The results are shown in Fig. 7. Jellyfish obtains substantially higher bisection bandwidth than LEGUP at each stage. In fact, by stage 2, Jellyfish has achieved higher bisection bandwidth than LEGUP in stage 8, meaning (based on each stage's cost) that Jellyfish builds an equivalent network at cost 60% lower than LEGUP.

A minority of these savings is explained by the fact that Jellyfish is more bandwidth-efficient than Clos networks, as exhibited by our earlier comparison with fat-trees. But in addition, LEGUP appears to pay a significant cost to enable it to incrementally-expand a Clos topology; for example, it leaves some ports unused in order to ease expansion in later stages. We conjecture that to some extent, this greater incremental expansion cost is fundamental to Clos topologies.

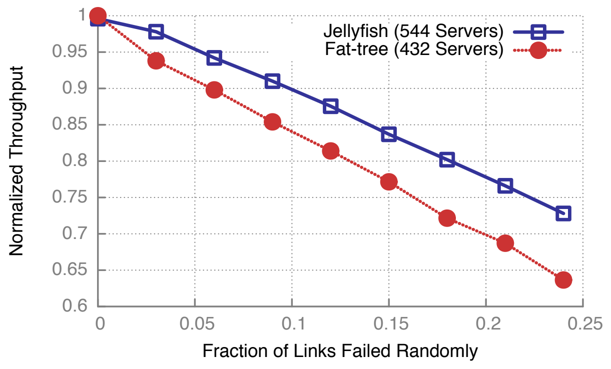


Figure 8: *Jellyfish is highly resilient to failures: Normalized throughput per server decreases more gracefully for Jellyfish than for a same-equipment fat-tree as the percentage of failed links increases. Note that the y-axis starts at* 60% *throughput; both topologies are highly resilient to failures.*

4.3 Failure Resilience

Jellyfish provides good path redundancy; in particular, an r-regular random graph is almost surely r-connected [9]. Also, the random topology maintains its (lack of) structure in the face of link or node failures – a random graph topology with a few failures is just another random graph topology of slightly smaller size, with a few unmatched ports on some switches.

Fig. 8 shows that the Jellyfish topology is even more resilient than the same-equipment fat-tree (which itself is no weakling). Note that the comparison features a fat-tree with fewer servers, but the same cost. This is to justify Jellyfish's claim of supporting a larger number of servers using the same equipment as the fat-tree, in terms of capacity, path length, and *resilience* simultaneously.

5 Routing & Congestion Control

While the above experiments establish that Jellyfish topologies have high capacity, it remains unclear whether this potential can be realized in real networks. There are two layers which can affect performance in real deployments: routing and congestion control. In our experiments with various combinations of routing and congestion control for Jellyfish (§5.1), we find that standard ECMP does not provide enough path diversity for Jellyfish, and to utilize the entire capacity we need to also use longer paths. We then provide in-depth results for Jellyfish's throughput and fairness using the best setting found earlier—k-shortest paths and multipath TCP (§5.2). Finally, we discuss practical strategies for deploying k-shortest-path routing (§5.3).

5.1 ECMP is not enough

Evaluation methodology: We use the simulator developed by the MPTCP authors for both Jellyfish and fat-tree. For routing, we test: (a) ECMP (equal cost multipath routing; We used 8-way ECMP, but 64-way ECMP does not perform much better, see Fig. 9), a standard strategy to distribute flows over shortest paths; and (b) k-shortest paths routing, which could be useful for Jellyfish because it can utilize longer-than-shortest paths. For k-shortest paths, we use Yen's Loopless-Path Ranking algorithm [2, 46] with $k = 8$ paths. For congestion control, we test standard TCP (1 or 8 flows per server pair) and the recently proposed multipath TCP (MPTCP) [44], using the recommended value of 8 MPTCP subflows. The traffic model continues to be a random permutation at the server-level, and as before, for the fat-tree comparisons, we build Jellyfish using the same switching equipment as the fat-tree.

Summary of results: Table 1 shows the average per server throughput as a percentage of the servers' NIC rate for two sample Jellyfish and fat-tree topologies under different routing and load balancing schemes. We make two observations: (1) ECMP performs poorly for Jellyfish, not providing enough path diversity. For random permutation traffic, Fig. 9 shows that about 55% of links are used by no more than 2 paths under ECMP; while for 8-shortest path routing, the number is 6%. Thus we need to make use of k-shortest paths. (2) Once we use k-shortest paths, each congestion control protocol works as least as well for Jellyfish as for the fat-tree.

The results of Table 1 depend on the oversubscription level of the network. In this context, we attempt to match fat-tree's performance given the routing and congestion control inefficiencies. We found that Jellyfish's advantage slightly reduces in this context compared to using idealized routing as before: In comparison to the same-equipment fat-tree (686 servers), now we can support, at same or higher performance, 780 servers (*i.e.*, 13.7% more that the fat-tree) with TCP, and 805 servers (17.3% more) with MPTCP. With ideal routing and congestion control, Jellyfish could support 874 servers (27.4% more). However, as we show quantitatively in §5.2, Jellyfish's advantage improves significantly with scale. At the largest scale we could simulate, Jellyfish supports 3,330 servers to the fat-tree's 2,662 — a $>$ 25% improvement (after accounting for routing and congestion control inefficiencies).

5.2 k-Shortest-Paths With MPTCP

The above results demonstrate using one representative set of topologies that using k-shortest paths with MPTCP yields higher performance than ECMP/TCP. In this sec-

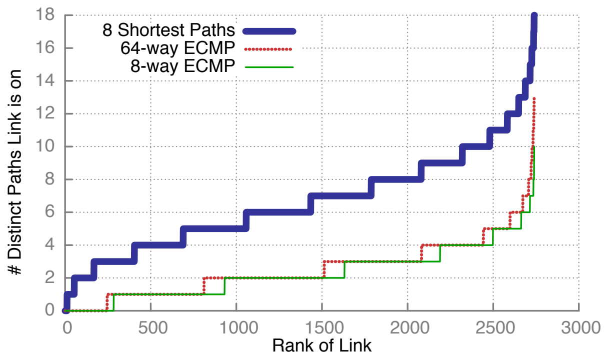


Figure 9: *ECMP does not provide path diversity for Jellyfish: Inter-switch link's path count in ECMP and k-shortest-path routing for random permutation traffic at the server-level on a typical Jellyfish of* 686 *servers (built using the same equipment as a fat-tree supports* 686 *servers). For each link, we count the number of distinct paths it is on. Each network cable is considered as two links, one for each direction.*

Congestion control	Fat-tree (686 svrs) ECMP	Jellyfish (780 svrs) ECMP	Jellyfish (780 svrs) 8-shortest paths
TCP 1 flow	48.0%	57.9%	48.3%
TCP 8 flows	92.2%	73.9%	92.3%
MPTCP 8 subflows	93.6%	76.4%	95.1%

Table 1: *Packet simulation results for different routing and congestion control protocols for Jellyfish (*780 *servers) and a same-equipment fat-tree (*686 *servers). Results show the normalized per server average throughput as a percentage of servers' NIC rate over* 5 *runs. We did not simulate the fat-tree with* 8*-shortest paths because ECMP is strictly better, and easier to implement in practice for the fat-tree.*

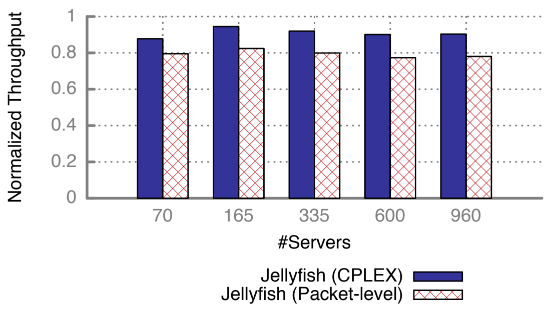


Figure 10: *Simple k-shortest path forwarding with MPTCP exploits Jellyfish's high capacity well: We compare the throughput using the same Jellyfish topology with both optimal routing, and our simple routing mechanism using MPTCP, which results in throughput between* 86% – 90% *of the optimal routing in each case. Results are averaged over* 10 *runs.*

tion we measure the efficiency of k-shortest path routing with MPTCP congestion control against the optimal performance (presented in §4), and later make comparisons against fat-trees at various sizes.

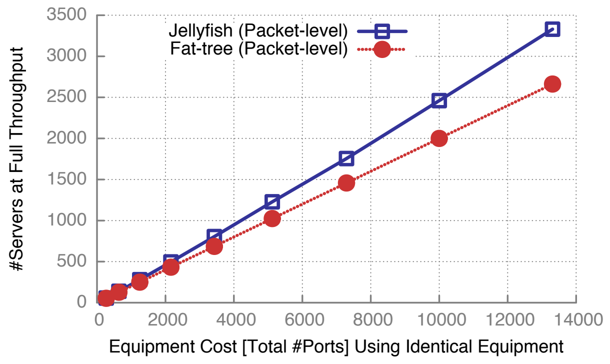


Figure 11: *Jellyfish supports a larger number of servers (>25% at the largest scale shown, with an increasing trend) than the same-equipment fat-tree at the same (or higher) throughput, even with inefficiencies of routing and congestion control accounted for. Results are averages over* 20 *runs for topologies smaller than* 1,400 *servers, and averages over* 10 *runs for larger topologies.*

Routing and Congestion Control Efficiency: The result in Fig. 10 shows the gap between the optimum performance, and the performance realized with routing and congestion control inefficiencies. At each size, we use the same slightly oversubscribed[10] Jellyfish topology for both setups. In the worst of these comparisons, Jellyfish's packet-level throughput is at ∼86% of the CPLEX optimal throughput. (In comparison, the fat-tree's throughput under MPTCP/ECMP is 93-95% of its optimum.) There is a possibility that this gap can be closed using smarter routing schemes, but nevertheless, as we discuss below, Jellyfish maintains most of its advantage over the fat-tree in terms of the number of servers supported at the the same throughput.

Fat-tree Throughput Comparison: To compare Jellyfish's performance against the fat-tree, we first find the average per-server throughput a fat-tree yields in the packet simulation. We then find (using binary search) the number of servers for which the average per-server throughput for the comparable Jellyfish topology is either the same, or higher than the fat-tree; this is the same methodology applied for Table 1. We repeat this exercise for several fat-tree sizes. The results (Fig. 11) are similar to those in Fig. 2(c), although the gains of Jellyfish are reduced marginally due to routing and congestion control inefficiencies. Even so, at the maximum scale of our experiment, Jellyfish supports 25% more servers than the fat-tree (3,330 in Jellyfish, versus 2,662 for the fat-tree). We note however, that even at smaller scale (for instance, 496 servers in Jellyfish, to 432 servers in the fat-tree) the improvement can be as large as ∼15%.

[10]An undersubscribed network may simply show 100% throughput, masking some of the routing and transport inefficiency.

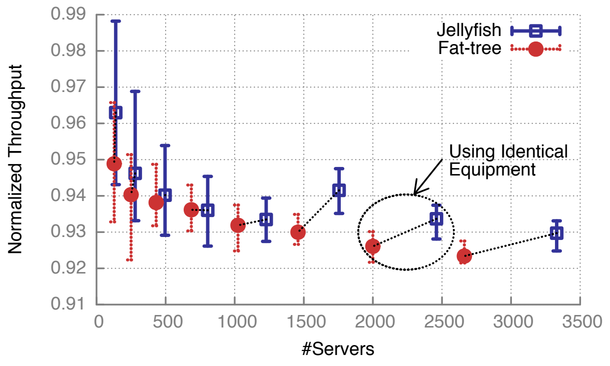


Figure 12: *The packet simulation's throughput results for Jellyfish show similar stability as the fat-tree. (Note that the y-axis starts at 91% throughput.) Average, minimum and maximum throughput-per-server values are shown. The data plotted is from the same experiment as Fig. 11. Jellyfish has the same or higher average throughput as the fat-tree while supporting a larger number of servers. Each Jellyfish data-point uses equipment identical to the closest fat-tree data-point to its left (as highlighted in one example).*

We show in Fig. 12, the stability of our experiments by plotting the average, minimum and maximum throughput for both Jellyfish and the fat-tree at each size, over 20 runs (varying both topologies and traffic) for small sizes and 10 runs for sizes >1,400 servers.

Fairness: We evaluate how flow-fair the routing and congestion control is in Jellyfish. We use the packet simulator to measure each flow's throughput in both topologies and show in Fig. 13, the normalized throughput per flow in increasing order. Note that Jellyfish has a larger number of flows because we make all comparisons using the same network equipment and the larger number of servers supported by Jellyfish. Both the topologies have similarly good fairness; Jain's fairness index [25] over these flow throughput values for both topologies: 0.991 for the fat-tree and 0.988 for Jellyfish.

5.3 Implementing *k*-Shortest-Path Routing

In this section, we discuss practical possibilities for implementing *k*-shortest-paths routing. For this, each switch needs to maintain a routing table containing for each other switch, *k* shortest paths.

OpenFlow [30]: OpenFlow switches can match end-to-end connections to routing rules, and can be used for routing flows along pre-computed *k*-shortest paths. Recently, Devoflow [35] showed that OpenFlow rules can be augmented with a small set of local routing actions for randomly distributing load over allowed paths, without invoking the OpenFlow controller.

SPAIN [36]: SPAIN allows multipath routing by using VLAN support in commodity off-the-shelf switches. Given a set of pre-computed paths, SPAIN merges these paths into multiple trees, each of which is mapped to a separate VLAN. SPAIN supports arbitrary topologies, and can enable use of *k*-shortest path routing in Jellyfish.

MPLS [40]: One could set up MPLS tunnels between switches such that all the pre-computed *k*-shortest paths between a switch pair are configured to have the same cost. This would allow switches to perform standard equal-cost load balancing across paths.

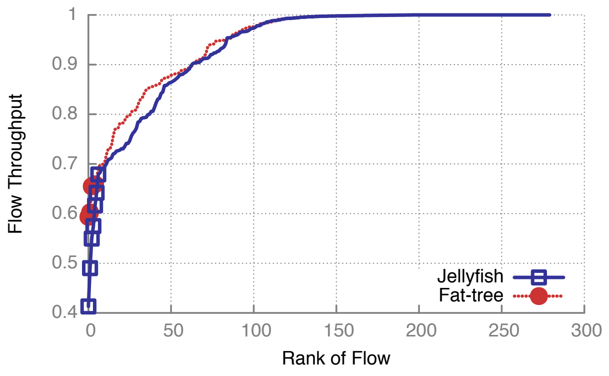


Figure 13: *Both Jellyfish and the fat-tree show good flow-fairness: The distribution of normalized flow throughputs in Jellyfish and fat-tree is shown for one typical run. After the few outliers (shown with points), the plot is virtually continuous (the line). Note that Jellyfish has more flows because it supports a higher number of servers (at same or higher* per-server *throughput). Jain's fairness index for both topologies is* ~99%.

6 Physical Construction and Cabling

Key considerations in data center wiring include:

- **Number of cables:** Each cable represents both a material and a labor cost.
- **Length of cables:** The cable price/meter is \$5-6 for both electrical and optical cables, but the cost of an optical transceiver can be close to \$200 [37]. We limit our interest in cable length to whether or not a cable is short enough, i.e., <10 meters in length [17, 26], for use of an electrical cable.
- **Cabling complexity:** Will Jellyfish awaken the dread spaghetti monster? Complex and irregular cabling layouts may be hard to wire and thus susceptible to more wiring errors. We will consider whether this is a significant factor. In addition, we attempt to design layouts that result in aggregation of cables in bundles, in order to reduce manual effort (and hence, expense) for wiring.

In the rest of this section, we first address a common concern across data center deployments: handling wiring errors (§6.1). We then investigate cabling Jellyfish in two deployment scenarios, using the above metrics (number, length and complexity of cabling) to compare against cabling a fat-tree network. The first deployment scenario is represented by small clusters (~1,000 servers); in this category we also include the intra-container clusters for 'Container Data Centers' (CDC)[11] (§6.2). The second deployment scenario is represented by massive-scale data centers (§6.3). In this paper we only analyze massive data centers built using containers, leaving more traditional data center layouts to future work.[12]

6.1 Handling Wiring Errors

We envision Jellyfish cabling being performed using a blueprint automatically generated based on the topology and the physical data center layout. The blueprint is then handed to workers to connect cables manually.

While some human errors are inevitable in cabling, these are easy to detect and fix. Given Jellyfish's sloppy topology, a small number of miswirings may not even require fixing in many cases. Nevertheless, we argue that fixing miswirings is relatively inexpensive. For example, the labor cost of cabling is estimated at ~10% of total cabling cost [37]. With a pessimistic estimate where the total cabling cost is 50% of the network cost, the cost of fixing (for example) 10% miswirings would thus be just 0.5% of the network cost. We note that wiring errors can be detected using a link-layer discovery protocol [32].

6.2 Small Clusters and CDCs

Small clusters and CDCs form a significant section of the market for data centers, and thus merit separate consideration. In a 2011 survey [15] of 300 US enterprises (with revenues ranging from \$1B-\$40B) which operate data centers, 57% of data centers occupy between 5,000 and 15,000 square feet; and 75% have a power load <2MW, implying that these data centers house a few thousand servers [13]. As our results in §4.1 show, even at a few hundred servers, cost-efficiency gains from Jellyfish can be significant (~20% at 1,000 servers). Thus, it is useful to deploy Jellyfish in these scenarios.

We propose a cabling optimization (along similar lines as the one proposed in [6]). The key observation is that in a high-capacity Jellyfish topology, there are more than twice as many cables running between switches than from servers to switches. Thus, placing all the switches in close proximity to each other reduces cable length, as well as manual labor. This also simplifies the small amounts of rewiring necessary for incremental expansion, or for fixing wiring errors.

[11] As early as 2006, The Sun Blackbox [3] promoted the idea of using shipping containers for data centers. There are also new products in the market exploiting similar physical design ideas [4, 5, 24].

[12] The use of container-based data centers seems to be an industry trend, with several players, Google and Microsoft included, already having container-based deployments [17].

Number of cables: Requiring fewer network switches for the same server pool also implies requiring fewer cables ($15-20\%$ depending on scale) than a fat-tree. This also implies that there is more room (and budget) for packing more servers in the same floor space.

Length of cables: For small clusters, and inside CDC containers using the above optimization, cable lengths are short enough for electrical cables without repeaters.

Complexity: For a few thousand servers, space equivalent to 3-5 standard racks can accommodate the switches needed for a full bisection bandwidth network (using available 64-port switches). These racks can be placed at the physical center of the data center, with aggregate cable bundles running between them. From this 'switch-cluster', aggregate cables can be run to each server-rack. With this plan, manual cabling is fairly simple. Thus, the nightmare cable-mess image a random graph network may bring to mind is, at best, alarmist.

A unique possibility allowed by the assembly-line nature of CDCs, is that of fabricating a random-connect *patch panel* such that workers only plug cables from the switches into the panel in a regular easy-to-wire pattern, and the panel's internal design encodes the random interconnect. This could greatly accelerate manual cabling.

Whether or not a patch panel is used, the problems of layout and wiring need to be solved only *once* at design time for CDCs. With a standard layout and construction, building automated tools for verifying and detecting miswirings is also a one-time exercise. Thus, the cost of any additional complexity introduced by Jellyfish would be amortized over the production of many containers.

Cabling under expansion: Small Jellyfish clusters can be expanded by leaving enough space near the 'switch-cluster' for adding switches as servers are added at the periphery of the network. In case no existing switch-cluster has room for additional switches, a new cluster can be started. Cable aggregates run from this new switch-cluster to all new server-racks and to all other switch-clusters. We note that for this to work with only electrical cabling, the switch-clusters need to be placed within 10 meters of each other as well as the servers. Given the constraints the support infrastructure already places on such facilities, we do not expect this to be a significant issue.

As discussed before, the Jellyfish expansion procedure requires a small amount of rewiring. The addition of every two network ports requires two cables to be moved (or equivalently, one old cable to be disconnected and two new cables to be added), since each new port will be connected to an existing port. The cables that need to be disconnected and the new cables that need to be attached can be automatically identified. Note that in the 'switch-cluster' configuration, all this activity happens at one location (or with multiple clusters, only between these clusters). The only cables not at the switch-cluster are the ones between the new switch and the servers attached to it (if any). This is just *one* cable aggregate.

We note that the CDC usage may or may not be geared towards incremental expansion. Here the chief utility of Jellyfish is its efficiency and reliability.

6.3 Jellyfish in Massive-Scale Data Centers

We now consider massive scale data centers built by connecting together multiple containers of the type described above. In this setting, as the number of containers grows, most Jellyfish cables are likely to be between containers. Therefore, inter-container cables in turn require expensive optical connectors, and Jellyfish can result in excessive cabling costs compared to a fat-tree.

However, we argue that Jellyfish can be adapted to wire massive data centers with lower cabling cost than a fat-tree, while still achieving higher capacity and accommodating a larger number of servers. For cabling the fat-tree in this setting, we apply the layout optimization suggested in [6], *i.e.,* make each fat-tree 'pod' a container, and divide the core-switches among these pods equally. With this physical structure, we can calculate the number of intra-container cables (from here on referred to as 'local') and inter-container cables ('global') used by the fat-tree. We then build a Jellyfish network placing the same number of switches as a fat-tree pod in a container, and using the same number of containers. The resulting Jellyfish network can be seen as a 2-layered random graph—a random graph within each container, and a random graph between containers. We vary the number of local and global connections to see how this affects performance in relation to the unrestricted Jellyfish network.

Note that with the same switching equipment as the fat-tree, Jellyfish networks would be overprovisioned if we used the same numbers of servers. To make sure that any loss of throughput due to our cable-optimization is clearly visible, we add a larger number of servers per switch to make Jellyfish oversubscribed.

Fig. 14 plots the capacity (average server throughput) achieved for 4 sizes[13] of 2-layer Jellyfish, as we vary the number of local and global connections, while keeping the total number of connections constant for a topology. Throughput is normalized to the corresponding unrestricted Jellyfish. The throughput drops by $<6\%$ when 60% of the network connections per switch are 'localized'. The percentage of local links for the equivalent fat-tree is 53.6%. Thus, Jellyfish can achieve a higher degree of localization, while still having a higher capacity network; recall that Jellyfish is 27% more efficient than the fat-tree at the largest scale (§4.1). The effect of cable localization on throughput was similar across the sizes we tested. For the fat-tree, the fraction of local links (conveniently given by $0.5(1+1/k)$ for a fat-tree built with k-port switches) *decreases* marginally with size.

[13]These are very far from massive scale, but these simulations are directed towards observing general trends. Much larger simulations are beyond our simulator's capabilities.

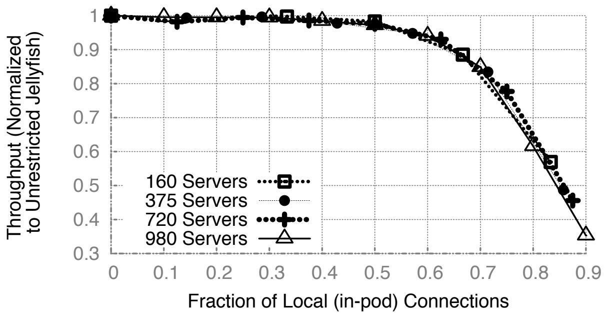


Figure 14: *Localization of Jellyfish random links is a promising approach to tackle cabling for massive scale data centers: As links are restricted to be more and more local, the network capacity decreases (as expected). However, when* 50% *of the random links for each switch are constrained to remain inside the pod, there is* $<3\%$ *loss of throughput.*

Complexity: Building Jellyfish over switches distributed uniformly across containers will, with high probability, result in cable assemblies between every pair of containers. A 100,000 server data center can be built with ~40 containers. Even if *all* ports (except those attached to servers) from each switch in each container were connected to other containers, we could aggregate cables between each container-pair leaving us with ~800 such cable assemblies, each with fewer than 200 cables. With the external diameter of a 10GBASE-SR cable being only $245um$, each such assembly could be packed within a pipe of radius $<1cm$. Of course, with higher over-subscription at the inter-container layer, these numbers could decrease substantially.

Cabling under expansion: In massive-scale data centers, expansion can occur through addition of new containers, or expansion of containers (if permissible). Laying out spare cables together with the aggregates between containers is helpful in scenarios where a container is expanded. When a new container is added, new cable aggregates must be laid out to every other container. Patch panels can again make this process easier by exposing the ports that should be connected to the other containers.

7 Conclusion

We argue that random graphs are a highly flexible architecture for data center networks. They represent a novel approach to the significant problems of incremental and heterogeneous expansion, while enabling high capacity, short paths, and resilience to failures and miswirings.

We thank Chandra Chekuri, Indranil Gupta, Gianluca Iannaccone, Steven Lumetta, Sylvia Ratnasamy, Marc Snir, and the anonymous reviewers for helpful comments; Andy Curtis for providing code for bisection bandwidth calculation, and sharing LEGUP topologies for comparison; and the MPTCP authors for sharing their simulator. This work was supported in part by National Science Foundation grant CNS 10-40396.

References

[1] CPLEX Linear Program Solver. http://www-01.ibm.com/software/integration/optimization/cplex-optimizer/.

[2] An implementation of k-shortest path algorithm. http://code.google.com/p/k-shortest-paths/.

[3] Project blackbox. http://www.sun.com/emrkt/blackbox/story.jsp.

[4] Rackable systems. ICE Cube modular data center. http://www.rackable.com/products/icecube.aspx.

[5] SGI ICE Cube Air expandable line of modular data centers. http://sgi.com/products/data_center/ice_cube_air.

[6] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. In *SIGCOMM*, 2008.

[7] L. N. Bhuyan and D. P. Agrawal. Generalized hypercube and hyperbus structures for a computer network. *IEEE Transactions on Computers*, 1984.

[8] B. Bollobás. The isoperimetric number of random regular graphs. *Eur. J. Comb.*, 1988.

[9] B. Bollobás. Random graphs, 2nd edition. 2001.

[10] B. Bollobás and W. F. de la Vega. The diameter of random regular graphs. In *Combinatorica 2*, 1981.

[11] A. Broder and E. Shamir. On the second eigenvalue of random regular graphs. In *FOCS*, 1987.

[12] F. Comellas and C. Delorme. The (degree, diameter) problem for graphs. http://maite71.upc.es/grup_de_grafs/table_g.html/.

[13] A. R. Curtis, T. Carpenter, M. Elsheikh, A. Lopez-Ortiz, and S. Keshav. REWIRE: an optimization-based framework for unstructured data center network design. In *INFOCOM*, 2012.

[14] A. R. Curtis, S. Keshav, and A. Lopez-Ortiz. LEGUP: using heterogeneity to reduce the cost of data center network upgrades. In *ACM CoNEXT*, 2010.

[15] Digital Reality Trust. What is driving the us market? http://goo.gl/qiaRY, 2001.

[16] Facebook. Facebook to expand Prineville data center. http://goo.gl/fJAoU.

[17] N. Farrington, G. Porter, S. Radhakrishnan, H. H. Bazzaz, V. Subramanya, Y. Fainman, G. Papen, and A. Vahdat. Helios: A hybrid electrical/optical switch architecture for modular data centers. In *SIGCOMM*, 2010.

[18] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. VL2: a scalable and flexible data center network. In *SIGCOMM*, 2009.

[19] C. Guo, G. Lu, D. Li, H. Wu, X. Zhang, Y. Shi, C. Tian, Y. Zhang, and S. Lu. BCube: A high performance, server-centric network architecture for modular data centers. In *SIGCOMM*, 2009.

[20] C. Guo, H. Wu, K. Tan, L. Shi, Y. Zhang, and S. Lu. DCell: a scalable and fault-tolerant network structure for data centers. In *SIGCOMM*, 2008.

[21] L. Gyarmati and T. A. Trinh. Scafida: A scale-free network inspired data center architecture. In *SIGCOMM CCR*, 2010.

[22] J. Hamilton. Datacenter networks are in my way. http://goo.gl/Ho6mA.

[23] HP. HP EcoPOD. http://goo.gl/8A0Ad.

[24] HP. Pod 240a data sheet. http://goo.gl/axHPp.

[25] R. K. Jain, D.-M. W. Chiu, and W. R. Hawe. A quantitative measure of fairness and discrimination for resource allocation in shared computer systems. Technical report, Digital Equipment Corporation, 1984.

[26] J. Kim, W. J. Dally, S. Scott, and D. Abts. Technology-driven, highly-scalable dragonfly topology. *ACM SIGARCH*, 2008.

[27] F. T. Leighton. Introduction to parallel algorithms and architectures: Arrays, trees, hypercubes. 1991.

[28] A. Licis. Data center planning, design and optimization: A global perspective. http://goo.gl/Sfydq.

[29] B. D. McKay and N. C. Wormald. Uniform generation of random regular graphs of moderate degree. *J. Algorithms*, 1990.

[30] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. OpenFlow: enabling innovation in campus networks. *SIGCOMM CCR*, 2008.

[31] A. B. Michael, M. Nolle, and G. Schreiber. A message passing model for communication on random regular graphs. In *International Parallel Processing Symposium (IPPS)*, 1996.

[32] Microsoft. Link layer topology discovery protocol. http://goo.gl/bAcZ5.

[33] R. Miller. Facebook now has 30,000 servers. http://goo.gl/EGD2D.

[34] R. Miller. Facebook server count: 60,000 or more. http://goo.gl/79J4.

[35] J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, A. R. Curtis, and S. Banerjee. DevoFlow: cost-effective flow management for high performance enterprise networks. In *Hotnets*, 2010.

[36] J. Mudigonda, P. Yalagandula, M. Al-Fares, and J. C. Mogul. SPAIN: COTS data-center ethernet for multipathing over arbitrary topologies. In *NSDI*, 2010.

[37] J. Mudigonda, P. Yalagandula, and J. Mogul. Taming the flying cable monster: A topology design and optimization framework for data-center networks. 2011.

[38] R. N. Mysore, A. Pamboris, N. Farrington, N. Huang, P. Miri, S. Radhakrishnan, V. Subramanya, and A. Vahdat. Portland: A scalable fault-tolerant layer 2 data center network fabric. In *SIGCOMM*, 2009.

[39] L. Popa, S. Ratnasamy, G. Iannaccone, A. Krishnamurthy, and I. Stoica. A cost comparison of datacenter network architectures. In *ACM CoNEXT*, 2010.

[40] E. Rosen, A. Viswanathan, and R. Callon. Multiprotocol Label Switching Architecture. RFC 3031, 2001.

[41] J.-Y. Shin, B. Wong, and E. G. Sirer. Small-world datacenters. *ACM Symposium on Cloud Computing (SOCC)*, 2011.

[42] A. Singla, A. Singh, K. Ramachandran, L. Xu, and Y. Zhang. Proteus: a topology malleable data center network. In *HotNets*, 2010.

[43] G. Wang, D. G. Andersen, M. Kaminsky, K. Papagiannaki, T. S. E. Ng, M. Kozuch, and M. Ryan. c-Through: Part-time optics in data centers. In *SIGCOMM*, 2010.

[44] D. Wischik, C. Raiciu, A. Greenhalgh, and M. Handley. Design, implementation and evaluation of congestion control for Multipath TCP. In *NSDI*, 2011.

[45] H. Wu, G. Lu, D. Li, C. Guo, and Y. Zhang. MDCube: a high performance network structure for modular data center interconnection. In *ACM CoNEXT*, 2009.

[46] J. Yen. Finding the k shortest loopless paths in a network. *Management Science*, 1971.

OSA: An Optical Switching Architecture for Data Center Networks with Unprecedented Flexibility

Kai Chen*, Ankit Singla†, Atul Singh‡, Kishore Ramachandran‡
Lei Xu‡, Yueping Zhang‡, Xitao Wen*, Yan Chen*
*Northwestern University, †UIUC, ‡NEC Labs America, Inc.

Abstract

Data center networks (DCNs) form the backbone infrastructure of many large-scale enterprise applications as well as emerging cloud computing providers. This paper describes the design, implementation and evaluation of OSA, a novel Optical Switching Architecture for DCNs. Leveraging runtime reconfigurable optical devices, OSA dynamically changes its topology and link capacities, thereby achieving unprecedented flexibility to adapt to dynamic traffic patterns. Extensive analytical simulations using both real and synthetic traffic patterns demonstrate that OSA can deliver high bisection bandwidth (60%-100% of the non-blocking architecture). Implementation and evaluation of a small-scale functional prototype further demonstrate the feasibility of OSA.

1 Introduction

Many on-line services, such as those offered by Amazon, Google, FaceBook, and eBay, are powered by massive data centers hosting hundreds of thousands of servers. The network interconnect of the data center plays a key role in the performance and scalability of these services. As the number of hosted applications and the amount of traffic grow, the industry is looking for larger server-pools, higher bit-rate network interconnects, and smarter workload placement approaches to satisfy the demand. To meet these goals, a careful examination of traffic characteristics, operator requirements, and network technology rends is critical.

Traffic characteristics. Several recent DCN proposals attempt to provide uniformly high capacity between all servers [2, 16, 17, 28]. Given that it is not known *a priori* which servers will require high speed connectivity, for a static, electrical network, this appears to be the only way to prevent localized bottlenecks. However, for many real scenarios, such a network may not be fully utilized at all times. For instance, measurement on a 1500-server Microsoft production DCN reveals that only a few ToRs are hot and most of their traffic goes to a few other ToRs [20]. Likewise, an analysis of high-performance computing applications shows that the bulk of inter-processor traffic is degree-bounded and slowly-changing [4]. Thus, even for a few thousand servers, uniformly high capacity networks appear to be an overkill. As the size of the network grows, this weighs on the cost, power consumption and complexity of such networks.

Dealing with the oversubscribed networks. Achieving high performance for data center services is challenging in the oversubscribed networks. One approach is to use intelligent workload placement algorithms to allocate network-bound service components to physical hosts with high bandwidth connectivity [19], *e.g.*, placing these components on the same rack. Such workloads exist in practice: dynamic creation and deletion of VM instances in Amazon's EC2 or periodic backup services running between an EC2 (compute) instance and an S3 (storage) bucket. An alternate approach is to flexibly allocate more network bandwidth to service components with heavy communications. If the network could "shape-shift" in such fashion, this could considerably simplify the workload placement problem.

Higher bit-rates. There is an increasing trend towards deploying 10 GigE NICs at the end hosts. In fact, Google already has 10 GigE deployments and is pushing the industry for 40/100 GigE [22, 24, 30]. Deploying servers with 10 GigE naturally requires much higher capacity at the aggregation layers of the network. Unfortunately, traditional copper-wire 10 GigE links are not viable for distances over 10 meters [15] due to their high power budget and larger cable size, necessitating the need to look for alternative technologies.

Optical networking technology is well suited to meet the above challenges. Optical network elements support on-demand provisioning of connectivity and capacity where required in the network, thus permitting the construction of thin, but flexible interconnects for large server pools. Optical links can support higher bit-rates over longer distances using less power than copper cables. Moreover, optical switches run cooler than electrical ones [11], resulting in lower heat dissipation and cheaper cooling cost. The long-term advantage of optics in DCNs has been noted in the industry [1, 11].

Recent efforts in c-Through [35] and Helios [15] provide a promising direction to exploit optical networking technology (*e.g.*, one-hop high-capacity optical circuits) for building DCNs. Following this trailblazing research, we present OSA, a novel Optical Switching Architecture for DCNs. OSA achieves high flexibility by leveraging and extending the techniques devised by previous work, and further combining them with novel techniques of its own. Similar to the previous work, OSA leverages reconfigurability of optical devices to dynamically set up one-hop optical circuits. Then, OSA employs the novel hop-by-hop stitching of multiple optical links to provide all-to-all connectivity for mice flows and bursty communications, and also to handle workloads involving high fan-in/out hotspots [18] that the existing one-hop electrical/optical architectures cannot address efficiently via their optical interconnects[1]. Further, OSA dynamically adjusts the capacities on the optical links to satisfy changing traffic demand at a finer granularity. Additionally, to make efficient use of expensive optical ports, OSA introduces Circulator (Sec. 2.2), a bi-directionality-enabling component for simultaneous transmission in both directions over the same circuit, which potentially doubles the usage of MEMS ports.

Overall, the highlights of this paper are as follows.

A flexible DCN architecture. Given a number N of Top-of-Rack (ToR) switches and a design-time-fixed parameter k, OSA can assume *any* k-regular topology over the N ToRs. To illustrate how many options this gives us, consider that for just N=20, there are over 12 billion (non-isomorphic) connected 4-regular graphs [25]. In addition, OSA allows the capacity of each edge in this k-regular topology to be varied from a few Gb/s to a few hundred Gb/s on-demand. Evaluation results in Sec. 5.2.2 suggest an up to 150% and 50% performance improvement brought by flexible topology and flexible link capacity, respectively.

An analysis of OSA-2560. We evaluate a particular instance of container-size OSA architecture, OSA-2560 (N=80, k=4), with 2560 servers via extensive simulations and analysis. Our evaluation results (Sec. 5.2) suggest that OSA-2560 can deliver high bisection bandwidth that is 60%-100% of the non-blocking network and outperform the hybrid structures by 80%-250% for both real and synthetic traffic patterns. Our analysis (Sec.3.3) shows that OSA incurs lower cost ($\sim$38%), lower ($\sim$37%) power consumption, and one order of magnitude simpler cabling complexity compared to a non-blocking Fattree [2] connecting a similar number of servers. Furthermore, compared with the hybrid structures, OSA has similar cost but consumes slightly less power. We believe that for data centers that expect skewed traffic demands, OSA provides a compelling tradeoff between cost, complexity and performance.

An implementation of OSA prototype. We build a small-scale 8-rack OSA prototype with real optical devices. Through this testbed, we evaluate the performance of OSA with all software and hardware overheads. Our results show that OSA can quickly adapt the topology and link capacities to meet the changing traffic patterns, and that it achieves nearly 60% of non-blocking bandwidth in the all-to-all communication. We further examine the impact of OSA design on bulk data transfer and mice flows, and find that the overhead introduced by hop-by-hop routing on mice flows is small: a 2 ms additional latency for a 7-hop routing with full background traffic. We also measure the device characteristics of the optical equipment, evaluate the impact of multi-hop optical-electrical-optical (O-E-O) conversion, and discuss our experience building and evaluating the OSA prototype.

Limitations. OSA, in its current form, has limitations. Small flows, especially those latency-sensitive ones, may incur non-trivial penalty due to reconfiguration delays ($\sim$10ms). While the fraction of such affected flows is small (Sec. 7), we propose multiple avenues to solve this challenge. The second challenge is to scale OSA from a container-size to a larger date center consisting of tens to hundreds of thousands of servers. This requires non-trivial efforts in both architecture and management design, and is left as part of our ongoing investigation. In this paper, we describe OSA that is designed to connect few thousands of servers in a container.

Roadmap. In Sec. 2, we discuss the idea of OSA's unprecedented flexibility, followed by background on optical technologies for OSA. Then we describe OSA architecture (Sec. 3) and its algorithm design (Sec. 4) in response to traffic patterns. In Sec. 5 and Sec. 6, we evaluate OSA via extensive simulations and implementation respectively. We discuss some design issues and related work to OSA in Sec. 7 before concluding in Sec. 8.

2 Motivation and Background

We first use a motivating example to show what kind of flexibility OSA delivers. Then, we introduce the optical

[1]In the optical part of the existing hybrid electrical/optical architectures, one ToR only connects to one other ToR at a time. While it can connect to different ToRs at different times, the switching latency would be around 10 ms. As we will introduce, in OSA, one ToR can connect to multiple ToRs simultaneously at a time, and more importantly, multi-hop connection exist between any pair of remote ToRs via hop-by-hop circuit stitching.

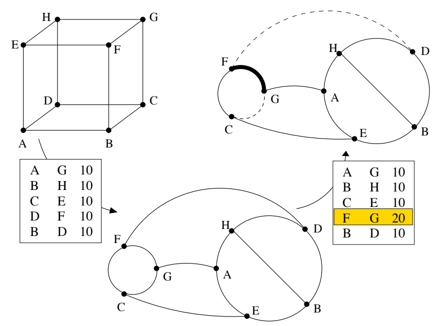


Figure 1: OSA adapts topology and link capacities to the changing traffic demands.

technologies that make OSA possible.

2.1 A Motivating Example

We discuss the utility of a flexible network using the simple hypothetical example in Fig. 1. On the left is a hypercube connecting 8 ToRs using 10G links. The traffic demand is shown in the bottom-left of Fig. 1. For this demand, no matter what routing paths are used on this hypercube, at least one link will be congested. One way to tackle this congestion is to reconnect the ToRs using a different topology (Fig. 1, bottom-center). In the new topology, all the communicating ToR pairs are directly connected and their demand can be perfectly satisfied.

Now, suppose the traffic demand changes (Fig. 1, bottom-right) with a new (highlighted) entry replacing an old one. If no adjustment is made, at least one link will face congestion. With the shortest path routing, $F \leftrightarrow G$ will be that link. In this scenario, one solution to avoid congestion is to increase the capacity of the $F \leftrightarrow G$ to 20G at the expense of decreasing capacity of link $F \leftrightarrow D$ and link $G \leftrightarrow C$ to 0. Critically, note that in all three topologies, the degree and the capacity of nodes remain the same, *i.e.*, 3 and 30G respectively.

As above, OSA's flexibility lies in its flexible topology and link capacity. In the absence of such flexibility, the above example would require additional links and capacities to handle both traffic patterns. More generally, a large variety of traffic patterns would necessitate 1:1 over-subscription (*i.e.*, non-blocking) network construction. OSA avoids the cost of constructing a non-blocking network while still providing equivalent performance for varying traffic patterns.

2.2 Optical Technologies

We now discuss the optical networking technologies that enable the above flexibility.

1. Wavelength Division Multiplexing (WDM): Depending on the channel spacing, using WDM, typically 40 or up to 100 channels or wavelengths can be transmitted over a single piece of fiber in the conventional or C-band. For the purposes of our architecture, each wavelength is rate-limited by the electrical port it connects to.

2. Wavelength Selective Switch (WSS): A WSS is typically a $1 \times N$ switch, consisting of one *common* port and N *wavelength* ports. It partitions (runtime-configurable within a few ms) the set of wavelengths coming in through the common port among the N wavelength ports. E.g., if the common port receives 80 wavelengths then it can route wavelengths 1–20 on port 1, wavelengths 30–40 and 77 on port 2, etc.

3. Optical Switching Matrix (OSM): Most OSM modules are bipartite $N \times N$ matrix where any input port can be connected to any one of the output ports. The most popular OSM technology uses Micro-Electro-Mechanical Switch, or MEMS. It can reconfigure to a new input/output matching within 10ms [33] by mechanically adjusting a microscopic array of mirrors. A few hundred ports are common for commercial products, and >1000 for research prototypes [14]. The current commercially available OSM modules are typically oblivious to the wavelengths carried across it. We use MEMS and OSM interchangeably.

4. Optical Circulators: Circulators enable bidirectional optical transmission over a fiber, allowing more efficient use of the ports of optical switches. An optical circulator is a three-port device: one port is a shared fiber or switching port, and the other two ports serve as send and receive ports.

5. Optical Transceivers: Optical transceivers can be of two types: coarse WDM (CWDM) and dense WDM (DWDM). We use DWDM-based transceivers, which support higher bit-rates and more wavelength channels in a single piece of fiber compared to CWDM.

3 OSA Architecture

In this section, we introduce how OSA[2] is built from the above described optical technologies. Our current design targets container-size DCNs.

3.1 Building Blocks

Flexible topology. OSA achieves flexible topology via exploiting the reconfigurability of MEMS. Say we start by connecting each of N ToRs to one port on an N-port MEMS. Given the MEMS bipartite port-matching, this implies that every ToR can only communicate with one other ToR at any instant, leaving the ToR level graph disconnected. If we connect N/k ToRs to k ports each at the MEMS, each ToR can communicate with k ToRs simultaneously. Here, $k > 1$ is the degree of a ToR, not

[2]We presented a preliminary design of OSA in an earlier workshop paper [32].

its port count, in the ToR graph. The configuration of the MEMS determines which set of ToRs are connected; and OSA must ensure that the ToR graph is connected when configuring the MEMS.

Given a ToR graph connected by optical circuits through the MEMS, we use *hop-by-hop* stitching of such circuits to achieve network-wide connectivity. To reach remote ToRs that are not directly connected, a ToR uses one of its k connections. This first-hop ToR receives the transmission over fiber, converts it to electrical signals, reads the packet header, and routes it towards the destination. At each hop, every packet is converted from optics to electronics and then back to optics (O-E-O) and switching at the ToR. Pure O-E-O conversion can be done in sub-nanoseconds [21]. Note that at any port, the aggregate transit, incoming and outgoing traffic cannot exceed the port's capacity in each direction. So, high-volume connections must use a minimal number of hops. OSA should manage the topology to adhere to this requirement. Evaluation in Sec. 6 quantifies the overhead (both O-E-O and switching) of hop-by-hop routing.

Flexible link capacity. Every ToR has degree k. If each edge had fixed capacity, multiple edges may need to be used for this ToR to communicate with another ToR at a rate higher than a single edge supports. To overcome this problem, OSA combines the capability of optical fibers to carry multiple wavelengths at the same time (WDM) with the dynamic reconfigurability of the WSS. Consequently, a ToR is connected to the MEMS through a multiplexer and a WSS unit.

Specifically, suppose ToR A wants to communicate with ToR B using w times the line speed of a single port. The ToR will use w ports, each associated with a (unique) wavelength, to serve this request. WDM enables these w wavelengths, together with the rest from this ToR, to be multiplexed into one optical fiber that feeds the WSS. The WSS splits these w wavelengths to the appropriate MEMS port which has a circuit to ToR B (doing likewise for $k-1$ other wavelengths). Thus, a $w\times$ (*line-speed*) capacity circuit is set up from A to B, at runtime. By varying the value of w for every MEMS circuit, OSA achieves dynamic capacity for every edge.

We note that a fiber cannot carry two channels over the same wavelength in the same direction. Moreover, to enable a ToR pair to communicate using all available wavelengths, we require that each ToR port (facing the optical interconnect) is assigned a wavelength unique across ports at this ToR. The same wavelength is used to receive traffic as well: each port thus sends and receives traffic at one fixed wavelength. The same set of wavelengths is recycled across ToRs. This allows all wavelengths at one ToR to be multiplexed and delivered after demultiplexing to individual ports at the destination ToR. This wavelength-port association is a design time decision.

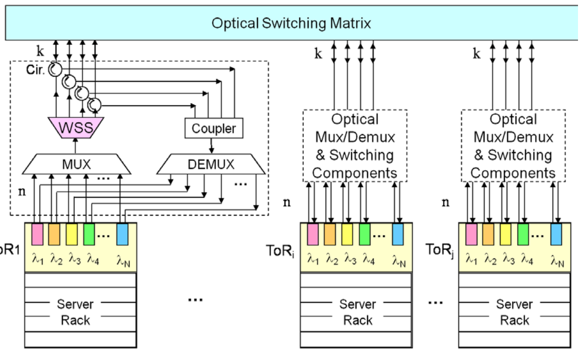

Optical Switching Matrix
k
Cir.
WSS
Coupler
MUX
DEMUX
n
ToR1
λ_1 λ_2 λ_3 λ_4 λ_N
Server Rack
Optical Mux/Demux & Switching Components
ToR$_i$
ToR$_j$

Figure 2: The overall OSA architecture; detailed structure is shown only for ToR$_1$ for clarity.

Efficient port usage. To make full use of the MEMS ports, we desire that each circuit over the MEMS be bidirectional. For this, we use optical circulators between the ToR and the MEMS ports. A circulator connects the send channel of the transceiver from a ToR to the MEMS (after the channel has passed through the WSS). It simultaneously delivers the traffic incoming towards a ToR from the MEMS, to this ToR. Note that even though the MEMS edges are bidirectional, the capacities of the two directions are independent of each other.

3.2 Putting it All Together: OSA-2560

Fig. 2 illustrates the general OSA architecture. We now discuss one specific instantiation, OSA-2560, with $N = 80$ ToRs, $W = 32$ wavelengths and ToR degree $k = 4$ using a 320-port MEMS to support 2560 servers.

Each ToR is a commodity electrical switch with 64 10-GigE ports [7]. 32 of these ports are connected to servers, while the remaining face the optical interconnect. Each port facing the optical interconnect has a transceiver associated with a fixed and unique wavelength for sending and receiving data. The transceiver uses separate fibers to connect to the send and receive infrastructures.

The send fiber from the transceivers from each of the 32 ports at a ToR is connected to an optical multiplexer. The multiplexer feeds a 1×4 WSS. The WSS splits the set of 32 wavelengths it sees into 4 groups, each group being transmitted on its own fiber. These fibers are connected to the MEMS via circulators to enable bidirectional communications. The 4 receive fibers from 4 circulators are connected to a power coupler (similar to a multiplexer, but simpler), which combines their wavelengths onto one fiber. This fiber feeds a demultiplexer, which splits each incoming wavelength to its associated port on the ToR.

We point out two key properties of the above interconnect. First, each ToR can communicate simultaneously with any 4 other ToRs. This implies that the MEMS configuration allows us to construct all possible 4-regular

Element	\$	W	Element	\$	W
ToR (10G port)	.5K[†]	12.5[†]	(DE)MUX	3K	0
MEMS	.5K[†]	0.24[†]	Coupler	.1K	0
WSS	1K[†]	1[†]	Circulator	.2K	0
Transceiver	.8K	3.5	-	-	-

Table 1: [†]Cost (USD) and power (Watt) per port for different elements, the values are from Helios [15].

graphs among ToRs. Second, through WSS configuration, the capacity of each of these 4 links can be varied in $\{0, 10, 20, \ldots, 320\}$ Gbps. The MEMS and WSS configurations are decided by a central OSA manager. The manager estimates the traffic demand, calculates appropriate configurations, and pushes them to the MEMS, WSS units and ToRs. This requires direct, out-of-band connections between the manager and these components. Our use of such a central OSA manager is inspired by many recent works [8, 15, 16, 28, 35] in the context of DCNs given that a DCN is usually owned and operated by a single organization.

Furthermore, we choose $k = 4$ for container-sized DCNs because it is a tradeoff between the network size and performance. A larger k value can enable one ToR to connect to more other ToRs simultaneously, thus achieving higher performance. However, given the fixed 320-port MEMS, it also means that fewer ToRs $(320/k)$ can be supported. Our experiments with $k = 1, 2, 4, 8$ indicate that $k = 4$ can deliver considerable bisection bandwidth between thousands of servers.

3.3 Analysis

Table 1 lists the cost and power usage of different network elements. Table 2 compares the traditional network, hybrid structure, OSA and Fattree.

Traditional over-subscribed network. For connecting 2560 servers using a two-tiered 2:1 oversubscribed architecture[3], we use 80 48×10G port ToR switches that have 32 ports connected to servers. The remaining 16 ports at each ToR are connected to aggregation switches. We use a total of 80 aggregation switches each with 16×10G ports. Note that the choice of 32 server-facing port and 16 aggregation-switch-facing ports results in 2:1 over-subscription. This architecture costs USD 4.6M and consumes 72.96KW. The number of cross-ToR fibers required is 1280. The bisection bandwidth provided is 50% of the non-blocking network. However, for skewed traffic demands, it is desirable to allocate a large fraction of this capacity to more demanding flows and achieve better cost/performance tradeoff.

[3]We picked 2:1 over-subscription ratio because, for all traffic patterns we studied, OSA delivers network bisection bandwidth that is at least 60% of the non-blocking network (Sec. 5). Thus, a 2:1 oversubscribed traditional network (50% of non-blocking) is a conservative comparison point.

Architecture	\$	KW	% of non-blocking
Traditional	4.6M	73	50%
Hybrid	5.6M	78	20%–50%[‡]
OSA	5.6M	73	60%–100%[‡]
Fattree	14.6M	196	100%

Table 2: Cost, power and performance for different network architectures to support 2560 servers with 10GigE ports. ([‡]For traffic patterns we evaluate in Sec. 5.)

Simplified model of the hybrid structure. Helios [15] and c-Through [35] are two well-known hybrid electrical/optical structures. The hybrid structure model we used here and in Sec. 5 is an abstract model that captures key aspects of both. In this model, each ToR has connections to an electrical network and an optical network. The electrical network is a two or three tiered tree with a certain over-subscription ratio (8:1 for Table 2). In the optical part, each ToR has only one optical link connecting to one other ToR, but this link is of unrestricted capacity. This hybrid structure costs USD 5.6M, consumes 78KW and has 480 long fibers – 160 above the MUX in optical part and 320 above the ToRs in electrical part.

OSA. The total cost is approximately USD 5.6M, with a power consumption of 73KW. ToRs and transceivers are responsible for a large portion of the cost and power budget. Compared to the traditional architecture, the additional cost is mainly due to (DE)MUX and WSS units. The number of long fibers required by OSA is small – 320 fibers above the circulator layer. The ToR to circulator connection is very short and can be packaged with the ToR. OSA's cost is similar to the hybrid structure but is $\sim$20% more expensive than the traditional structure[4], however, it can dynamically adjust the bandwidth allocated to demanding flows. For all traffic demands we evaluated in Sec. 5, this enables OSA to achieve 60%-100% of the non-blocking bisection bandwidth. The power consumption is nearly identical to that of the traditional oversubscribed network; this is because the total number of electrical ports used in both architectures are identical, and optical components add negligible power.

Fattree. The cost and power of Fattree depends solely on the number of ports needed: a Fattree topology with p port Ethernet switches can connect $p^3/4$ hosts with a total of $5{*}p^3/4$ ports. Note that for 10G port electrical switches, optical transceiver for remote connection is a necessity. To connect 2560 servers, Fattree costs 14.6M USD. The power consumption is 196KW. The number of fibers required above the ToR layer is 5120. Fattree is more expensive and consumes more power, because it

[4]We also note here that the cost of optics is expected to fall significantly with commoditization and production volume. Much of these benefits have already been reaped for electrical technology. There is also scope for packaging multiple components on a chip - the 32 transceivers and the MUX could be packaged into one chip. This will reduce power consumption, cost, as well as the number of fibers.

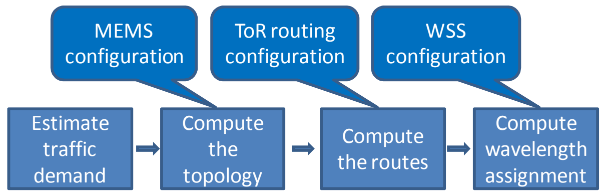


Figure 3: The steps in OSA control algorithm.

is designed to provide non-blocking connectivity and is also highly fault-tolerant. Our intention is not to perform a head-to-head comparison with Fattree, but to illustrate the cost/power/performance tradeoff of building a non-blocking network architecture.

Summary. For data center deployments where skewed traffic demands are expected, we believe that OSA is a better alternative than both Fattree and traditional over-subscribed networks: Fattree suffers from significantly higher cost and cabling complexity, and traditional architectures are inflexible and cannot assign spare bandwidth to demanding flows on the fly. Compared with the hybrid structure, OSA can achieve better performance with similar cost and power consumption.

4 Design

In this section, we present OSA network optimization in detail. Our goal is to compute the optimal topology and link capacities such that the network bisection bandwidth is maximized for a given traffic demand. In this paper we do not focus on estimating traffic demand, which can be achieved by following similar techniques presented in [15, 18, 35]. For optimization, we seek to find: 1) a MEMS configuration to adjust the topology to localize high traffic volumes, 2) routes between ToRs to achieve high throughput, low latency or avoid congestion, and 3) a configuration for each WSS to provision the capacities of its outgoing links. As we show in [9], this optimization problem can be formulated as a mixed integer program, which is well known to be NP-hard. We next introduce an approximation solution.

4.1 Solution

We decompose the problem into three steps as shown in Fig. 3, *i.e.*, computing the topology, the routing and the wavelength assignment. In this paper, we adopt the traffic demand estimation method introduced by Hedera [3], which is based on the max-min fair bandwidth allocation for TCP flows in an ideal non-blocking network.

1. Compute the topology: We localize high-volume communicating ToR pairs over direct MEMS circuit links. This is accomplished by using a weighted b-matching [27], where b represents the number of ToRs that a ToR connects to via MEMS ($b = 4$ in OSA-2560). In the ToR graph, we assign the edge-weight between two ToRs as the estimated demand between them, and then cast the problem of localizing high-volume ToR connections to b-matching. Weighted b-matching is a graph theoretic problem for which polynomial-time algorithm exists [27]. We implement it using multiple perfect matchings, for which public library is available [23].

The b-matching graph above is not necessarily a connected graph. Fortunately, connectivity is easy to achieve via the edge-exchange operation [29]. First, we find all the connected components. If the graph is not connected, we select two edges $a{\rightarrow}b$ and $c{\rightarrow}d$ with lowest weights in different connected components, and connect them via replacing links $a{\rightarrow}b$ and $c{\rightarrow}d$ with links $a{\rightarrow}c$ and $b{\rightarrow}d$. We make sure that the links removed are not themselves cuts in the graph.

2. Compute the routes: Once we have connectivity, the MEMS configuration is known. We proceed to compute routes using any of the standard routing schemes such as the shortest path routing or low congestion routing. Note that some of the routes are single-hop MEMS connection while others are multi-hop ones. For simplicity, we use the shortest path routing in this paper. However, our framework can be readily applied to other routing schemes.

3. Compute the wavelength assignment: Given the traffic demand and routes between any pair of ToRs, we can easily compute the capacity desired on each ToR link in order to serve the traffic demand on this link.

With the desired capacity demand on each link, we need to provision a corresponding amount of wavelengths to serve the demand. However, wavelength assignment is not arbitrary: due to the contention, a wavelength can only be assigned to a ToR at most once. Given this constraint, we reduce the problem to an edge-coloring problem on a multigraph. We represent our ToR level graph as a multigraph. Multiple edges correspond to the number of wavelengths between two nodes, and we assume each wavelength has a unique color. Thus, a feasible wavelength assignment is equivalent to an assignment of colors to the edges of the multigraph so that no two adjacent edges have the same color – exactly the edge-coloring problem [12]. Edge-coloring is a known problem and fast heuristics are known [26]. Libraries implementing this are publicly available.

We also require at least one wavelength to be assigned to each edge on the physical topology. This guarantees an available path between any ToR-pair, which may be required for mice/bursty flows.

All the above steps are handled by the OSA manager. Specifically, the OSA manager interacts with MEMS, WSS units and ToRs to control the topology, link capacities and routing respectively. We note that our decomposition heuristic is not optimal and there is room to improve. However, it provides satisfactory gains as we will see.

5 Simulation

In this section, we evaluate OSA-2560 via analytical simulations. We start with the simulation methodology, and then present the results.

5.1 Simulation Methods

Simulation goals: Since our testbed only has 8 ToRs (Sec. 6), to evaluate OSA's capabilities at its intended scale, we conduct analytical estimation of network bisection bandwidth of OSA-2560 under various traffic patterns. Our results in this section are essentially computations of the expected bisection bandwidth in the steady state, ignoring software and hardware overheads which are considered in our testbed experiments in Sec. 6. We compare OSA with a non-blocking network, a hybrid network with varied over-subscription ratios in the electrical part and a 2:1 oversubscribed traditional network.

Communication patterns: We use the following real measurement traces and synthetic traffic data to evaluate the performance of OSA in the presence of changing communication patterns and traffic demands.

1. Mapreduce-demand: We collected real traffic matrices in a production data center with around 400 servers, which mainly runs Mapreduce applications[5]. We compute demands by averaging the traffic over 30-second periods. For each demand, we identify the communication pattern by filtering out mice flows and focusing on the elephant ones. We map these communication patterns onto OSA-2560 using spatial replication.

2. Measurement-based: Recent measurements [6, 18] reveal several data center traffic characteristics. One important feature is that hotspot ToR links are often associated with a high fan-in (or fan-out), and most of the traffic (80%) are within the rack, resulting in highly skewed distribution. We synthesize this kind of traffic patterns by randomly choosing 12 hotspots out of 80 racks, with each one connecting to 6-10 other randomly chosen ToRs respectively. We intentionally assume all traffic exit the rack in order to create intensive communications.

3. ToR Level Shifting: We index the ToR switches from 0 to 79 and shift traffic round-by-round. Initially, all servers in ToR i talk to all servers in ToRs $(i \pm 1)$ mod 80 and $(i \pm 2)$ mod 80. Then we shift these communications to servers in the next ToR after each round.

4. Server Level Shifting: We index the servers from 0 to 2559. We start with server i talking to 4 other servers: $(i \pm 32)$ mod 2560 and $(i \pm 64)$ mod 2560. With 32 servers in a rack, initially, this implies that each rack communicates with 4 other racks. In successive rounds, server i talks to $(i \pm (32+s))$ mod 2560 and $(i \pm (64+s))$ mod 2560 ($s = 4, 8, 12, \cdots$). This implies that each rack communicates with 6 racks in most rounds, with traffic spread across these 6 connections increasing and decreasing periodically.

5. Random Shifting: In each round, each server in ToR i talks to servers in up to 10 randomly selected ToRs. In this pattern, many ToRs may simultaneously talk to one ToR, creating hotspots and communication bottlenecks.

6. Increasing Destinations: We gradually increase the number of destinations for each ToR from 4 through 79 (*i.e.*, all-to-all communications) to further investigate the impact of traffic spread on OSA performance.

Evaluation metrics: First, we evaluate the network bisection bandwidth provided by OSA for each communication pattern. Then, we quantify the impact of flexible topology and flexible link capacity within OSA architecture respectively. Finally, we measure time cost of the control algorithm described in Sec 4.1. The experiments were conducted on a Dell Optiplex machine with Intel 2.33 GHz dual-core CPU and 4 GB Memory.

The hybrid structure: We simulate the hybrid structure model introduced in Sec. 3.3 which captures the key features of c-Through and Helios. To optimize the network to traffic, we run maximum weighted matching to determine which optical circuits to establish. Then we calculate how much of the remaining demand can be satisfied by the electrical network at best.

Traditional 2:1 oversubscribed network: We also simulate a 2:1 over-subscribed electrical network whose details were described earlier in Sec. 3.3.

5.2 Evaluation Results

5.2.1 Performance of OSA

In this experiment, the topology and link capacities are adaptively adjusted to the current traffic pattern. As soon as traffic pattern changes, the network reconfigures its topology instantaneously. In practice, the performance of OSA would be also impacted by the time taken to estimate the traffic demand, the time taken by the algorithms to identify the appropriate topology, and reconfiguration time of optical devices. Experimental results from our prototype will encompass these overheads (see Sec. 6).

Fig. 4 shows the average network bisection bandwidth over 100 instances of each traffic pattern obtained by different DCN structures. Note that all the results are normalized by the bisection bandwidth of the non-blocking scenario. We make following observations.

[5]The name of the production data center company is anonymized.

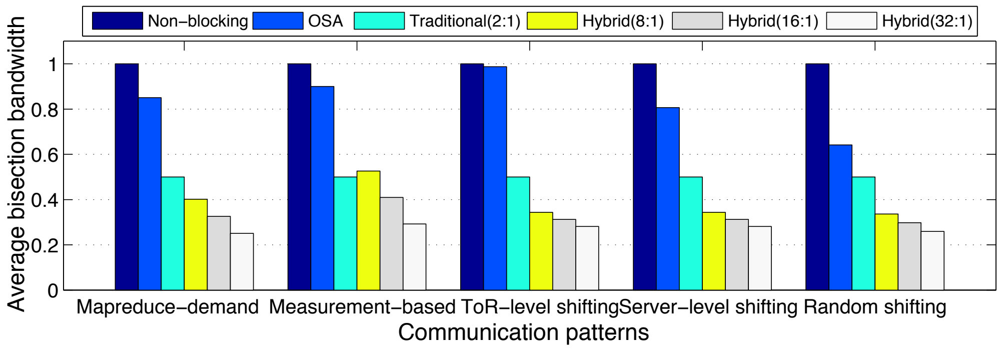


Figure 4: The average network bisection bandwidth (normalized) achieved for different communication patterns.

First, we find that OSA delivers high bisection bandwidth (60%-100% of non-blocking) for both real and synthetic traffic patterns. Under the Mapreduce-demand, OSA can provide over 80% of the non-blocking bandwidth. This is because OSA adaptively changes its topology and link capacities according to the present traffic pattern. In our simulation setting, we choose 4-regular graph for OSA. Because some ToRs talk to more than 4 (up to 8) other ToRs, OSA cannot assign direct circuits to feed all these communications. The multi-hop routing possibly causes congestion on the intermediate switches, leading to performance degradation. That is why OSA is 20% from non-blocking. From the figure, we find that OSA delivers higher bandwidth (90% of non-blocking) for the measurement-based pattern, because it has relatively less hotspots compared to the previous one.

Second, when each ToR communicates with 4 other ToRs (in the ToR-level shifting pattern), OSA achieves bisection bandwidth nearly identical to that of the non-blocking network. This result is not surprising given that OSA allows a 4-regular graph and hence provides 4 optical circuits at each ToR to perfectly support the demand. Note that the traditional 2:1 oversubscribed network delivers 50% of non-blocking for all traffic patterns.

Third, in our results (not shown here due to lack of space), we observe that the bisection bandwidth achieved by OSA oscillates periodically from approximately 60% to 100% (with average at 80%) of non-blocking for the server-level shifting pattern. This is because each ToR would periodically communicate with 4 and 6 other ToRs in such traffic pattern. We further observe that the bisection bandwidth obtained by OSA in the random shifting pattern is the worst – 60% of non-blocking. This is expected since the number of peers each ToR communicates with is larger than the other two shifting patterns. Specifically, for the ToR-level shifting, a ToR talks to 4 other peers; For the server-level shifting, a ToR communicates with 4-6 peers; While for the random shifting pattern, a ToR communicates with 5-20 peers. As discussed above, when the number of communication peers for a ToR is larger than 4, some flows will necessarily use multi-hop paths causing performance degradation. Concretely, for the ToR-level shifting most paths are direct, for the server-level shifting most paths are direct or 2 hops, and for the random shifting most paths are increased to 2-6 hops. The multi-hop paths taken by some flows contend for the available bandwidth at intermediate switches, thus limiting the peak achievable bandwidth.

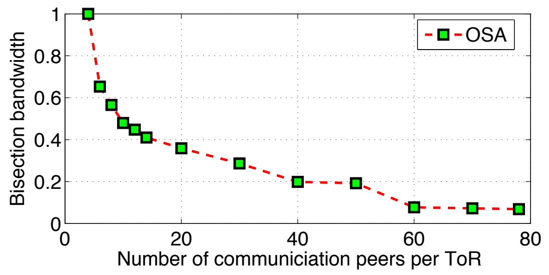
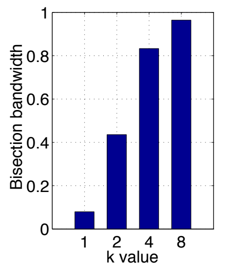


Figure 5: Network bisection bandwidth with an increasing number of peers each ToR communicates with.

Next, we present the bisection bandwidth achieved by OSA with an increasing number of inter-ToR communications. As it moves gradually to all-to-all communication (Fig. 5), as expected, the network bisection bandwidth drops due to extensive bandwidth contention at the ToRs. Note that the traditional 2:1 oversubscribed network would continue to perform at 50% of non-blocking. This result is presented only for comparison purposes since OSA is not designed for all-to-all communication.

Furthermore, we note that OSA outperforms the hybrid model by 80%-250% in our evaluation. This is not a surprising result because the hybrid model only has a perfect matching between ToRs in the optical part. This means that one ToR is able to talk to one other ToR at a time. We increase over-subscription ratios in the electrical part from 32:1 to 8:1 and see only incremental improvement due to the oversubscribed network. In contrast, in OSA-2560, we have a 4-regular graph meaning one ToR can directly communicate with 4 other ToRs simultaneously. Further, OSA also dynamically adapts its link capacities to the traffic demand. The higher flexibility of OSA leads to its better performance.

In Fig. 6, we inspect the performance delivered by



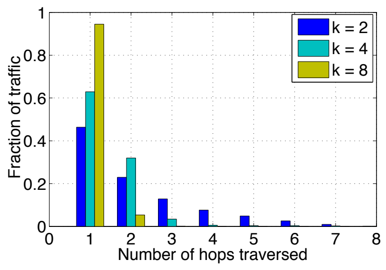


Figure 6: The performance of OSA with varied k values (left) and the number of hops traversed by traffic (right).

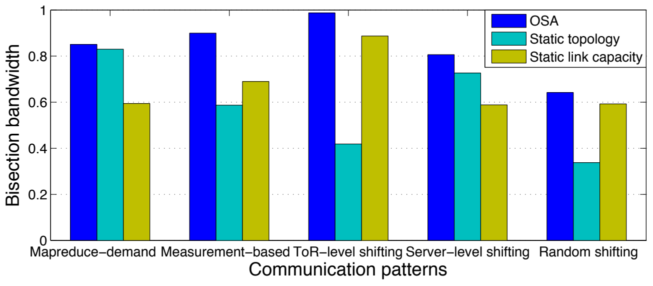


Figure 7: Effect of flexible topology and link capacity.

OSA with varied k values (left) and the number of hops traversed by the traffic (right) using the Mapreduce-demand. We assume that there are always 80 ToRs. It is evident from the left figure that with a larger k value, the network bisection bandwidth delivered is higher. However, the larger k value also necessitates more MEMS ports in order to support the same number of ToRs and servers. Note that $k = 2$, where we see low performance, is exactly equivalent to the optical part of the hybrid structure. From the right figure, we find that, for our case of OSA-2560 (*i.e.*, $k = 4$), the vast majority of traffic only traverses less than 3 hops - over 60% of traffic goes one hop and over 30% of traffic goes two hops. We also find that with a small k value, a considerable portion of traffic needs to traverse multiple hops to reach the destinations. When k increases, more traffic will go fewer hops, indicating better network performance. Though not shown, the similar trends hold for the rest traffic patterns.

5.2.2 Effect of Flexible Topology & Link Capacity

We quantify the effect of flexible topology and flexible link capacity respectively. For this purpose, in the first experiment we randomly select a fixed topology (*e.g.*, the one generated by the first instance of a traffic pattern), and only adjust the link capacity according to the current traffic pattern. In the second experiment, we hypothetically assume each link has 8 fixed wavelengths assigned (thus static link capacity), and only adjust the topology based on the current traffic pattern. Fig. 7 shows the bisection bandwidth of both scenarios and the original OSA. Comparing the static topology scenario with OSA, we observe up to $\frac{100\%-40\%}{40\%} = 150\%$ improvement due to the effect of flexible topology in case of the ToR-level shifting pattern. Comparing the static link capacity scenario with OSA, we observe up to $\frac{90\%-60\%}{60\%} = 50\%$ improvement because of the effect of flexible link capacity in case of the measurement-based traffic pattern. These results suggest that the flexible topology and link capacity are essential to improve the performance of OSA.

Control algorithm	Time (ms)
Estimate traffic demand	161
Compute the topology	48
Compute the routes	41
Compute the wavelength assignment	40
Total	290

Table 3: Time consumption of the control algorithm.

5.2.3 Time Cost of Control Algorithm

We measure the time cost of the OSA control algorithm as described in Sec 4.1. We run our current software implementation with 50 randomly selected traffic patterns that we used above and compute the average values for each step. As shown in Table 3, the total time is 290 ms. We observe that out of the 4 steps, traffic demand estimation is dominant (161 ms). The reason is that the algorithm for this step is based on the number of servers, while the rest are based on the number of ToRs. Note that our demand estimation algorithm is adopted directly from Hedera [3], which has recently been shown to be less than 100 ms for large data centers via parallelization over multiple cores or machines. This means there is a large room to speed up with advanced technologies.

Though most of the remaining steps take only tens of milliseconds, we still believe optimizations are possible throughout the control software to make it more responsive even for larger networks. For instance, b-matchings for 1,024 nodes could be computed in as few as 250 ms in the year 2000 with contemporary hardware [27]. It is also likely that better-performing, faster heuristics can be built based on more accurate models of the traffic.

6 Implementation

We built an OSA prototype with real optical devices. We next present the testbed setup and experiment results.

6.1 Testbed Setup

The testbed includes 8 Dell Optiplex servers, each emulating a virtual rack (V-Rack) of 4 virtual-machines (VMs). We use Xen hypervisor and CentOS 5.3 for Dom0 and DomU.We emulate the ToR switches using 4 Dell PowerEdge servers, each equipped with an Intel

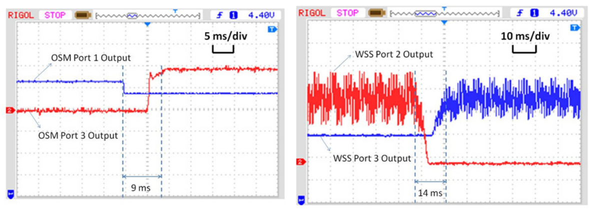


Figure 8: Switching time of our OSM and WSS.

2.4GHz quad-core CPU, 8GB DRAM and 12GigE NICs. Each server runs 2 VMs, giving us a total of 8 virtual ToRs (V-ToRs). Each V-ToR binds to 6 NICs: one connected to one V-Rack, one used for a control connection to OSA manager, and the remaining 4 used as upstream links to reach other V-ToRs via optical elements.

On top of each V-ToR is a 1×4 CoAdna WSS, a coupler, a circulator, a 1×4 MUX and DEMUX pair, and 4 transceivers (which are packaged into a media converted (MC) unit). As in Fig 2, each ToR uplink is connected to a transceiver, with the send-fiber of the transceiver connected through the MUX, the WSS and the circulator to the OSM; and the receive-fiber connected to the same circulator through the coupler and the DEMUX. We use a 1 Polatis series-1000 OSM with 32 ports which allows a 16×16 bipartite interconnect. (Each V-ToR has 2 uplinks connected to each of these two sets of 16 ports.) We use 4 wavelengths: 1545.32, 1544.53, 1543.73 and 1542.94 nm, corresponding to channel 40, 41, 42 and 43 of ITU grid with 100 GHz channel spacing.

Further, in our testbed, OSA manager is a separate Linux server and talks to the OSM and ToRs via Ethernet ports, and to the WSS units via RS-232 serial ports.

6.2 Understanding the Optical Devices

Two critical optical devices in OSA are OSM and WSS. A common concern for them is the reconfiguration overhead. To measure the overhead, Fig. 8 (left) shows the output power level on two ports of the OSM over time, during a reconfiguration event. We see a clear transition period, *i.e.*, from the high$\rightarrow$low output power level shift on one port, to the low$\rightarrow$high output power level shift on the other port. We observe that the switching delay of our OSM is 9 ms, consistent with [15, 35].

Next, we measure the reconfiguration time of WSS by switching a wavelength channel between two output ports. As shown in Fig. 8 (right), this transition period is around 14 ms. However, the reconfiguration of the OSM and WSS can be performed in parallel.

6.3 Understanding the O-E-O Conversion

To measure the impact of O-E-O conversion, we specially connect 4 servers as in Fig. 12 (left). Two servers in the middle are configured as routers and equipped with optical media converters. We create a routing loop by configuring the IP forwarding tables of the routers. In each router, we deploy a `netfilter` kernel module and utilize the `NF_IP_PRE_ROUTING` hook to intercept all IP packets. We record the time lag between the instant when the packets first arrive in the network and when their TTL expires. This way, we are able to measure the multi-hop latency for O-E-O conversion and compare it with the baseline where all servers are directly connected using only electrical devices. Results in Fig. 12 (right) compare the average one-hop switching latency for both the hybrid optical/electrical and pure electrical architectures under different traffic load. It is evident from the figure, that O-E-O conversion does not incur noticeable (the maximum deviation in the absolute value and standard deviation is 38 and 58 μs, respectively), if any, additional switching latency. demonstrating feasibility of O-E-O employed by OSA.

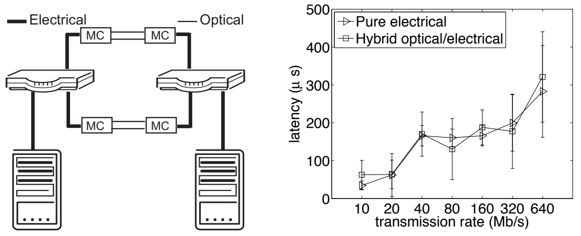


Figure 12: O-E-O conversion.

6.4 OSA System Performance

We conduct two sets of experiments: one is for original OSA and the other is OSA with static topology. We use synthetic traffic patterns similar to Sec 5.1. More specifically, traffic is described by parameters (t, r): servers in ToR i $(i = 0 \cdots 7)$ send traffic to servers in t ToRs, *i.e.*, $[i+r, i+r+1, ..., i+r+(t-1)]$. We change t from 1 to 7 to generate different traffic loads (t=7 means all-to-all communication). For each t, we vary r from 1 to 7.

Our goal is to compare the achieved bisection bandwidth of OSA against that of a non-blocking network as the traffic spread out (with increasing t), and to measure the effect of topology reconfiguration. Note that varying r with a fixed t does not produce fundamentally different traffic distributions, it merely permutes which ToRs talk with which other ToRs, thus necessitating a change of topology without a change in traffic load or spread.

In our testbed, the NICs of servers support 10, 100, and 1000 Mbps full-duplex modes. In all our experiments, we limit the maximum sending rate of each flow to be 100 Mbps. This enables us to emulate a non-blocking network for comparison: for OSA, all the uplink ports of ToRs are set at 100 Mbps, while for non-blocking, we increase particular uplink ports to 1000

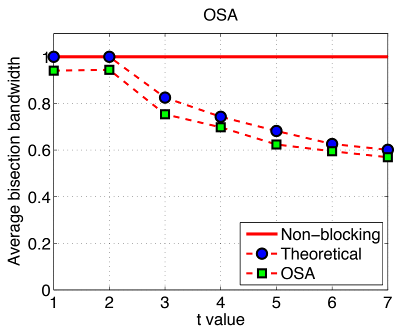


Figure 9: Average bisection bandwidth of OSA.

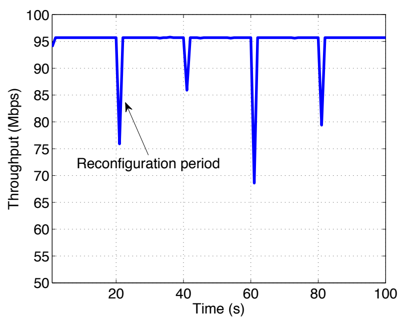


Figure 10: Throughput of a flow in the presence of reconfigurations.

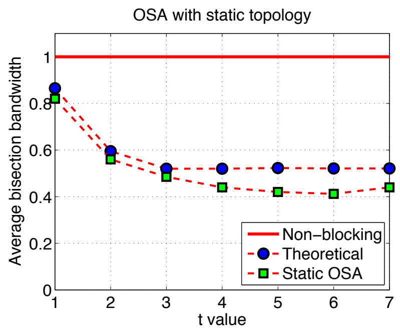


Figure 11: Average bisection bandwidth of OSA with a static topology.

Mbps to satisfy the traffic demands we simulate.

Results of OSA: Fig. 9 shows the average bisection bandwidth of OSA with changing traffic (t=$1 \cdots 7$). For each t, r steps 1 through 7 every 20 seconds. The network topology is dynamically reconfigured according to the current traffic demand. The results are along expected lines. We observe that the achieved bisection bandwidth of OSA is within 95% of the non-blocking network when t is 1 or 2. This is because when $t = 1$ each ToR talks with 2 other ToRs and when $t = 2$ each ToR talks with 4 other ToRs. Given that our topology is a 4-regular graph, OSA assigns direct links to each pair of communicating ToRs for efficient communication. For $t > 2$, the performance of OSA decreases, along similar lines as in the simulation (Sec. 5). A careful reader will notice that the performance of our testbed under all-to-all communication is 58% of non-blocking, much higher than that in our simulation results. The reason is simple: our testbed has 8 ToRs each with degree 4, while our simulations used a sparse graph with 80 ToRs each having degree 4. Our intention with the testbed results is to demonstrate the feasibility of OSA rather than to show the performance achieved in a real deployment.

Next, Fig. 10 shows the impact of optical device reconfigurability on the end-to-end throughput between two hosts. We observe that the performance drops during reconfiguration but quickly resumes after it finishes.

Finally, we also present the theoretical bisection bandwidth achievable in our testbed that ignores the overhead of reconfiguration, software routing, and TCP/IP protocol, etc. We observe that the gap between theoretically achievable values and OSA is within 5-7%, suggesting that our prototype is efficient.

Results of OSA with a static topology: We randomly select a topology and run the same experiments as above and present results in Fig. 11. Given the small diameter of our topology, the static topology OSA still achieves satisfactory performance. For example, in the worst case of all-to-all traffic (*i.e.*, $t = 7$), static OSA achieves more than 40% of the non-blocking network's bisection bandwidth. Since all the paths are 1 or 2-hop long, even the randomly selected topology performs satisfactorily.

For different t values, we find that the performance of OSA on the static topology is lower than that on dynamic topology by 10%-40%. This is because the topology is not optimized for the current traffic patterns. We expect that on a larger network where OSA topology is sparse (*e.g.*, the one we used in Sec. 5), this performance gap will become more pronounced, highlighting the need for a dynamically optimized network for better performance.

6.5 Bulk Data Transfer

We study how the network reconfiguration and multi-hop routing affect the bulk data transfer, *i.e.*, elephant flows.

Impact of network reconfiguration: We periodically reconfigure the network and observe the completion time of transferring a chunk of data (100 MB file transferred using `scp`) during the reconfiguration events. We present mean value of 100 trials. Fig. 13 shows our results and the baseline performance where no reconfiguration takes place. The stability time is defined as the lifetime for a single static topology, after which the network is reconfigured. We notice that the completion time increases in the presence of reconfigurations. After analyzing the network trace using `tcpdump`, we observed that the round trip time (RTT) and accordingly the initial retransmission time out (RTO) value in data centers are very small (sub-ms level), while network reconfiguration requires tens of ms (see Fig. 8). As a consequence, each reconfiguration almost always triggers RTO events, after which TCP waits for 200 ms (Linux default RTO value) before the next retransmission, thereby degrading throughput and increasing latency. Recent work [10, 34, 36] has pointed out TCP's RTO issues in data centers, and proposed to reduce it to the μs level by employing fine-grained timers. We expect TCP's performance in OSA under network reconfiguration to significantly improve once these changes are adopted. We also notice from the figure that the completion time decreases as the stability time increases – larger stability period re-

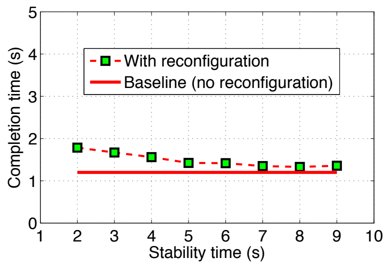


Figure 13: Impact of topology reconfiguration on bulk data transfer.

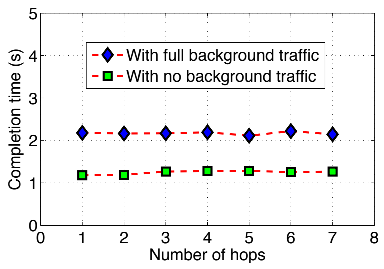


Figure 14: Impact of multi-hop routing on bulk data transfer.

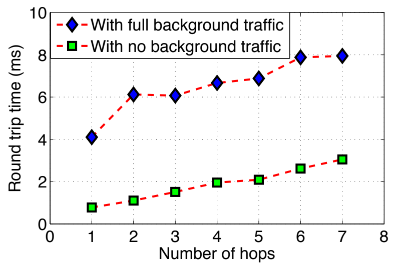


Figure 15: Impact of multi-hop routing on simulated mice flows.

sults in fewer network state changes and therefore fewer RTO events during the course of data transfer.

Impact of multi-hop routing: Our prototype topology is a low-diameter network due to a 8-node 4-regular graph. In order to evaluate the impact of multi-hop routing on bulk data transfer, we intentionally rearrange our 8 ToR switches in a line to form a linear topology with larger diameter. In Fig. 14, we measure the completion time of data transfer (100 MB file transferred using `scp`) in terms of the number of hops they pass through. Specifically, we consider two scenarios. In the first case, the network is free of background traffic. In the second case, all the links in the network are saturated by other elephant TCP flows. From the figure, we find that in both cases the completion time is relatively consistent regardless of the hops. This gives us confidence that multi-hop routing does not affect the performance of bulk data transfer seriously. We can further notice from the figure that the influence of multi-hop O-E-O conversion during data transfer is negligible. We also observe a nearly constant gap between the two curves, which is due to different link utilizations in the two experiments.

6.6 Mice Flow Transfer

After inspecting the performance of bulk data transfer, we further check the impact of multi-hop routing on transferring mice flows. For this purpose, we use `ping` to emulate latency sensitive flows and evaluate its performance with/without background traffic as above. Fig. 15 shows the average round trip time (RTT) of 100 `ping` packets with varying path lengths. As expected, the RTT increases with more hops. However, we find that the absolute increment is small: 1ms (without background traffic) and 2ms (with full background traffic), respectively, after 7 hops. These results suggest that the hop-by-hop stitching of optical paths is a feasible approach to provide overall connectivity. We note that network reconfiguration may have non-trivial impact on the latency-sensitive flows transfer, since it happens on the order of 10ms. We further discuss options to handle such issue in Sec. 7.

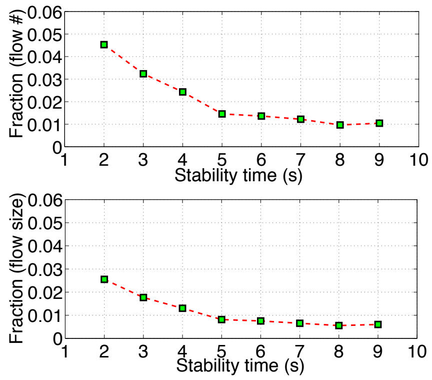


Figure 16: The potentially affected mice flows.

7 Discussion and Related Work

7.1 Mice Flow during Reconfiguration

OSA ensures that all ToRs are in a connected graph and uses hop-by-hop stitching of existing circuits to provide overall network connectivity. However, during network reconfiguration, a pair of ToRs may be temporarily disconnected for around 10 ms. While this can be largely tolerated by latency-insensitive applications such as Mapreduce or Dryad, it would affect those operating with latency-sensitive mice flows like Dynamo [13].

In Fig. 16, we estimate, in the worst case, how many mice flows (in terms of flow count and size) can be potentially affected due to the reconfiguration. We used the production data center traffic from Sec. 5.1 and used 10 MB to differentiate elephant flows from small ones. We find that for this particular dataset, when the stability time varies from 9 to 2 seconds, there are 1% to 4.5% of the mice flows that can be affected during the reconfigurations. This implies that as the network experiences more frequent reconfigurations, a larger fraction of mice flows may get affected. We next discuss two possible options to handle this issue.

Our basic idea is to reserve a static, connected channel in the OSA network. To do so, we can reserve a small number of wavelengths and MEMS/WSS ports that are never reconfigured and mice flows are always sent over them. Such a channel can be simply a spanning tree or other connected topologies. Given the topology of the channel is controlled by MEMS, we can arrange it in a low-diameter manner so that the transmission of mice flows is optimized. However, this approach consumes expensive MEMS/WSS ports, which otherwise can be better utilized for other applications or at stable time.

An alternative approach to building the channel without using MEMS/WSS ports is directly connecting all the ToRs together to form a ring or a star network. For the ring, we can reserve 2 ports on each ToR and directly connect them iteratively. In case of OSA-2560 with 80 ToRs, the diameter is 40 hops. To reduce the path length, it is possible to reserve more ports on each ToR and connect them structurally using DHT techniques [31], *e.g.*, the diameter is expected to be 3-4 hops with high probability for 80 ToRs if we reserve 4 ports on each ToR. Another option is to employ one additional central electrical switch – each ToR uses 1 port to connect to the central switch. Note that, in Helios or c-Through, the electrical switches (usually forming tree or even multi-root tree) are used for overall connectivity among all Pods/ToRs. In OSA, the all-to-all connectivity is maintained by optical components. A comprehensive evaluation and comparison of these solutions is part of our ongoing work.

7.2 OSA Applicability vs Traffic Properties

For all-to-all traffic, the non-oversubscribed network is indeed more appreciated. However, such workloads are neither reflected in our dataset nor in measurements elsewhere [6, 16, 18]. Our flexible OSA architecture would work best when traffic pattern is skewed and stable on the order of seconds. It has been noted in [20] over measurements of a 1500-server production DCN that "Only a few ToRs are hot and most of their traffic goes to a few other ToRs." Another study [16], also on a 1500-server production DCN, shows that more than 90% of bytes flow in elephant flows. Regarding traffic stability, a similarly sized study [5] shows that 60% of ToR-pairs see less than 20% change in traffic demand for between 1.6 to 2.2 seconds on average. Despite these, we expect that OSA may exhibit undesirable performance degradation if the traffic pattern is highly dynamic, in which case any topology adaptation mechanism may be unsuitable as the situation changes instantaneously. In practice, the infrastructure manager should choose the proper sensitivity of OSA according to the operational considerations.

7.3 Scalability

The current OSA design focuses on container-size DCNs. To scale, we may confront several challenges. The first one is the MEMS's port density. While the 1000-port MEMS is theoretically feasible, the largest MEMS as of today has 320 ports. One natural way to increase the port density is via interconnecting multiple small MEMS switches. However, this poses additional requirement for fast coordinated circuit switching. Secondly, larger network size necessitates more control and management. In our OSA-2560 with 80 ToRs, all the intelligences, *e.g.*, network optimization and routing, are handled by OSA manager. How to handle such tasks in a larger DCN with thousands of ToRs is an open question especially when the network environment is dynamic. Further, circuit visit delay [35] is another issue to notice when scaling. We are considering all these challenges in our continuous effort designing a scalable optical DCN.

7.4 Closely Related Work

OSA's design goals are closely related to those of c-Through [35] and Helios [15]. In both approaches, flows requiring high bandwidth are dynamically provisioned on optical circuits while a parallel electrical network is used to provide overall connectivity. OSA differs from these prior proposals in its degree of flexibility and its architecture. Both Helios and c-Through achieve some topology flexibility via a limited number of single-hop optical links. OSA can assume an arbitrary topology from a large class of k-regular connected graphs, while simultaneously allowing dynamic link capacities. Furthermore, unlike these architectures, OSA avoids using electrical components other than the ToR switches.

OSA is more comparable to c-Through than Helios, because its current target is inter-rack DCNs with a few thousand servers unlike Helios' inter-container mega-DCN scale. Qualitatively, OSA provides more flexibility than either Helios or c-Through and is able to serve a larger space of skewed traffic demands with performance similar to that of non-blocking interconnects. We present a coarse quantitative comparison with an abstract hybrid architecture model in Sec. 5, showing that OSA achieves significantly higher bisection bandwidth.

Recently, Kandula et al. [18, 20] proposed dynamically configuring 60GHz short-distance multi-Gigabit wireless links between ToRs to provide additional bandwidth for hotspots. Optical and wireless interconnects provide different trade-offs. For example, wired optical interconnects can deliver much more bandwidth at lower power usage over long distance, while wireless has lower costs and is easier to deploy though management and interference are challenging issues to deal with.

8 Conclusion

In this paper, we presented OSA, a novel Optical Switching Architecture for DCNs. OSA is highly flexible because it can adapt its topology as well as link capacities to different traffic patterns. We have evaluated OSA via extensive simulations and prototype implementation. Our results suggest that OSA can deliver high bisection bandwidth (60%-100% of non-blocking) for a series of traffic patterns. Our implementation and evaluation with the OSA prototype further demonstrate its feasibility.

Acknowledgements

We thank CoAdna Photonics and Polatis for the equipment support of WSS and OSM respectively, Guangxun Liao for helpful discussions on WSS properties, and Don DePaolo for help during the testbed setup. We thank our shepherd Dave Andersen, the anonymous reviewers, and Chuanxiong Guo for their valuable comments.

References

[1] ADC, "40 & 100 Gigabit Ethernet: An Imminent Reality," "http://www.adc.com/Attachment/1270718303886/108956AE,0.pdf".

[2] M. Al-Fares, A. Loukissas, and A. Vahdat, "A Scalable, Commodity Data Center Network Architecture," in *Proc. SIGCOMM*, 2008.

[3] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat, "Hedera: Dynamic Flow Scheduling for Data Center Networks," in *Proc. NSDI*, 2010.

[4] K. Barker and etal., "On the feasibility of optical circuit switching for high performance computing systems," in *Proc. SC*, 2005.

[5] T. Benson, A. Anand, A. Akella, and M. Zhang, "The Case for Fine-grained Traffic Engineering in Data-centers," in *Proc. USENIX INM/WREN*, 2010.

[6] T. Benson, A. Akella, and D. Maltz, "Network Traffic Characteristics of Data Centers in the Wild," in *Proc. IMC*, 2010.

[7] Broadcom, "BCM56840 Series Enables Mass Deployment of 10GbE in the Data Center," http://www.broadcom.com/products/features/BCM56840.php.

[8] K. Chen, C. Guo, H. Wu, J. Yuan, Z. Feng, Y. Chen, S. Lu, and W. Wu, "Generic and Automatic Address Configuration for Data Center Networks," in *Proc. SIGCOMM*, 2010.

[9] K. Chen, A. Singla, A. Singh, K. Ramachandran, L. Xu, Y. Zhang, X. Wen, and Y. Chen, "OSA: An Optical Switching Architecture for Data Center Networks with Unprecedented Flexibility," Northwestern University, Tech. Rep., 2012.

[10] Y. Chen, R. Griffith, J. Liu, R. H. Katz, and A. D. Joseph, "Understanding TCP Incast Throughput Collapse in Datacenter Networks," in *Proc. ACM WREN*, 2009.

[11] CIR, "40G Ethernet C- Closer Than Ever to an All-Optical Network," "http://cir-inc.com/resources/40-100GigE.pdf".

[12] Edge coloring. [Online]. Available: http://en.wikipedia.org/wiki/Edge_coloring.

[13] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, and P. Vosshall, "Dynamo: Amazon's Highly Available Key-value Store," in *Proc. SOSP*, 2007.

[14] J. K. et.al, "1100×1100 port MEMS-based optical crossconnect with 4-dB maximum loss," *IEEE Photonics Technology Letters*, vol. 15, no. 11, pp. 1537 –1539, 2003.

[15] N. Farrington, G. Porter, S. Radhakrishnan, H. H. Bazzaz, V. Subramanya, Y. Fainman, G. Papen, and A. Vahdat, "Helios: A Hybrid Electrical/Optical Switch Architecture for Modular Data Centers," in *Proc. ACM SIGCOMM*, 2010.

[16] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta, "VL2: A Scalable and Flexible Data Center Network," in *Proc. ACM SIGCOMM*, 2009.

[17] C. Guo, G. Lu, D. Li, H. Wu, X. Zhang, Y. Shi, C. Tian, Y. Zhang, and S. Lu, "BCube: A High Performance, Server-centric Network Architecture for Modular Data Centers," in *Proc. ACM SIGCOMM*, 2009.

[18] D. Halperin, S. Kandula, J. Padhye, P. Bahl, and D. Wetherall, "Augmenting Data Center Networks with Multi-gigabit Wireless Links," in *Proc. SIGCOMM*, 2011.

[19] J. Hamilton, "Data Center NetworksAre in My Way," "http://mvdirona.com/jrh/TalksAndPapers/JamesHamilton_CleanSlateCTO2009.pdf".

[20] S. Kandula, J. Padhye, and P. Bahl, "Flyways To De-Congest Data Center Networks," in *Proc. ACM HotNets*, 2009.

[21] G. Keeler, D. Agarwal, C. Debaes, B. Nelson, N. Helman, H. Thienpont, and D. Miller, "Optical pump-probe measurements of the latency of silicon CMOS optical interconnects," *IEEE Photonics Technology Letters*, vol. 14, no. 8, pp. 1214 – 1216, 2002.

[22] C. Lam, H. Liu, B. Koley, X. Zhao, V. Kamalov, and V. Gill, "Fiber optic communication technologies: What's needed for datacenter network operations," 2010.

[23] LEMON Library. [Online]. Available: http://lemon.cs.elte.hu.

[24] H. Liu, C. F. Lam, and C. Johnson, "Scaling Optical Interconnects in Datacenter Networks Opportunities and Challenges for WDM," in *Proc. IEEE Symposium on High Performance Interconnects*, 2010.

[25] M. Meringer, "Regular Graphs," http://www.mathe2.uni-bayreuth.de/markus/reggraphs.html.

[26] J. Misra and D. Gries, "A constructive proof of Vizing's Theorem," *Inf. Process. Lett.*, vol. 41, no. 3, pp. 131–133, 1992.

[27] M. Müller-Hannemann and A. Schwartz, "Implementing weighted b-matching algorithms: insights from a computational study," *J. Exp. Algorithmics*, vol. 5, p. 8, 2000.

[28] R. N. Mysore, A. Pamboris, N. Farrington, N. Huang, P. Miri, S. Radhakrishnan, V. Subramanya, and A. Vahdat, "PortLand: A Scalable Fault-Tolerant Layer 2 Data Center Network Fabric," in *Proc. ACM SIGCOMM*, 2009.

[29] K. Obraczka and P. Danzig, "Finding Low-Diameter, Low Edge-Cost, Networks," USC, Tech. Rep., 1997.

[30] J. Rath, "Google Eyes "Optical Express" for its Network," "http://www.datacenterknowledge.com/archives/2010/05/24/google-eyes-optical-express-for-its-network/".

[31] A. Rowstron1 and P. Druschel, "Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems," in *Proc. Middleware*, 2001.

[32] A. Singla, A. Singh, K. Ramachandran, L. Xu, and Y. Zhang, "Proteus: A Topology Malleable Data Center Network," in *Proc. ACM HotNets*, 2010.

[33] T. Truex, A. A. Bent, and N. W. Hagood, "Beam steering optical switch fabric utilizing piezoelectric actuation technology," in *Proc. NFOEC*, 2003.

[34] V. Vasudevan and etal., "Safe and Effective Fine-grained TCP Retransmissions for Datacenter Communication," in *Proc. ACM SIGCOMM*, 2009.

[35] G. Wang, D. G. Andersen, M. Kaminsky, K. Papagiannaki, T. S. E. Ng, M. Kozuch, and M. Ryan, "c-Through: Part-time Optics in Data Centers," in *Proc. ACM SIGCOMM*, 2010.

[36] H. Wu, Z. Feng, C. Guo, and Y. Zhang, "ICTCP: Incast Congestion Control for TCP," in *Proc. ACM CoNext*, 2010.

Less is More: Trading a little Bandwidth for Ultra-Low Latency in the Data Center

Mohammad Alizadeh, Abdul Kabbani[†], Tom Edsall[⋆], Balaji Prabhakar, Amin Vahdat[†§], and Masato Yasuda[¶]
Stanford University [†]Google [⋆]Cisco Systems [§]U.C. San Diego [¶]NEC Corporation, Japan

Abstract

Traditional measures of network goodness—goodput, quality of service, fairness—are expressed in terms of bandwidth. Network latency has rarely been a primary concern because delivering the highest level of bandwidth essentially entails driving up latency—at the mean and, especially, at the tail. Recently, however, there has been renewed interest in latency as a primary metric for mainstream applications. In this paper, we present the HULL (High-bandwidth Ultra-Low Latency) architecture to balance two seemingly contradictory goals: near baseline fabric latency and high bandwidth utilization. HULL leaves 'bandwidth headroom' using Phantom Queues that deliver congestion signals before network links are fully utilized and queues form at switches. By capping utilization at less than link capacity, we leave room for latency sensitive traffic to avoid buffering and the associated large delays. At the same time, we use DCTCP, a recently proposed congestion control algorithm, to adaptively respond to congestion and to mitigate the bandwidth penalties which arise from operating in a bufferless fashion. HULL further employs packet pacing to counter burstiness caused by Interrupt Coalescing and Large Send Offloading. Our implementation and simulation results show that by sacrificing a small amount (e.g., 10%) of bandwidth, HULL can dramatically reduce average and tail latencies in the data center.

1 Introduction

For decades, the primary focus of the data networking community has been on improving overall network goodput. The initial shift from circuit switching to packet switching was driven by the bandwidth and hardware inefficiencies of reserving network resources for bursty communication traffic. The Transmission Control Protocol (TCP) [26] was born of the need to avoid bandwidth/congestion collapse in the network and, subsequently, to ensure bandwidth fairness [14, 39, 45] among the flows sharing a network. Discussion to add quality-of-service capability to the Internet resulted in proposals such as RSVP [55], IntServ [10] and DiffServ [35], which again focussed on bandwidth provisioning.

This focus on bandwidth efficiency has been well justified as most Internet applications typically fall into two categories. Throughput-oriented applications, such as file transfer or email, are not sensitive to the delivery times of individual packets. Even the overall completion times of individual operations can vary by multiple integer factors in the interests of increasing overall network throughput. On the other hand, latency-sensitive applications—such as web browsing and remote login—are sensitive to per-packet delivery times. However, these applications have a human in the loop and completion time variations on the order of hundreds of milliseconds or even seconds have been thought to be acceptable, especially in the interests of maintaining high average bandwidth utilization. Hence, we are left with a landscape where the network is not optimized for latency or the predictable delivery of individual packets.

We are motivated by two recent trends that make it feasible and desirable to make low latency communication a primary metric for evaluating next-generation networks.
Data centers. A substantial amount of computing, storage, and communication is shifting to data centers. Within the confines of a single building—characterized by low propagation delays, relatively homogeneous equipment, and a single administrative entity able to modify software protocols and even influence hardware features—delivering predictable low latency appears more tractable than solving the problem in the Internet at large.
Ultra-low latency applications. Several applications and platforms have recently arisen that necessitate very low latency RPCs; for example, high-frequency trading (see [30]), high-performance computing, and RAMCloud [37, 38]. These applications are characterized by a request–response loop involving machines, not humans, and operations involving multiple parallel requests/RPCs

to thousands of servers. Since an operation completes when all of its requests are satisfied, the tail latency of the individual requests are required to be in microseconds rather than in milliseconds to maintain quality of service and throughput targets. As platforms like RAMCloud are integrated into mainstream applications such as social networking, search and e-commerce, they must share the network with throughput-oriented traffic which consistently moves terabytes of data.

There are several points on the path from source to destination at which packets currently experience delay: end-host stacks, NICs (network interface cards), and switches. Techniques like kernel bypass and zero copy [44, 12] are significantly reducing the latency at the end-host and in the NICs; for example, 10Gbps NICs are currently available that achieve less than 1.5μs per-packet latency at the end-host [41].

In this paper, we consider the latency in the network switching nodes. We propose HULL (for High-bandwidth Ultra-Low Latency), an architecture for simultaneously delivering predictable ultra-low latency and high bandwidth utilization in a shared data center fabric. The key challenge is that high bandwidth typically requires significant in-network buffering while predictable, ultra-low latency requires essentially no in-network buffering. Considering that modern data center fabrics can forward full-sized packets in microseconds (1.2μs for 1500 bytes at 10Gbps) and that switching latency at 10Gbps is currently 300–500ns [19, 23], a one-way delivery time of 10μs (over 5 hops) is achievable across a large-scale data center, *if queueing delays can be reduced to zero*. However, given that at least 2MB of on-chip buffering is available in commodity 10Gbps switches [19] and that TCP operating on tail-drop queues attempts to fully utilize available buffers to maximize bandwidth, one-way latencies of up to a few milliseconds are quite possible—and have been observed in production data centers.[1] This is a factor of 1,000 increase from the baseline. Since the performance of parallel, latency-sensitive applications are bound by tail latency, these applications must be provisioned for millisecond delays when, in fact, microsecond delays are achievable.

Our observation is that it is possible to reduce or eliminate network buffering by marking congestion based not on queue occupancy (or saturation) but rather based on the utilization of a link approaching its capacity. In essence, we cap the amount of bandwidth available on a link in exchange for significant reduction in latency.

Our motivation is to trade the resource that is relatively plentiful in modern data centers, i.e., bandwidth, for the resource that is both expensive to deploy and results in substantial latency increase—buffer space. Data center switches usually employ on-chip (SRAM) buffering to keep latency and pin counts low. However, in this mode, even a modest amount of buffering takes about 30% of the die area (directly impacting cost) and is responsible for 30% of the power dissipation. While larger in size, off-chip buffers are both more latency intensive and incur a significantly higher pin count. The references [5, 25] describe the cost of packet buffers in high bandwidth switching/routing platforms in more detail. The above considerations indicate that higher bandwidth switches with more ports could be deployed earlier if fewer chip transistors were committed to buffers.

The implementation of HULL centers around Phantom Queues, a switch mechanism closely related to existing virtual queue-based active queue management schemes [21, 31]. Phantom queues simulate the occupancy of a queue sitting on a link that drains at *less than* the actual link's rate. Standard ECN [40] marking based on the occupancy of these phantom queues is then used to signal end hosts employing DCTCP [3] congestion control to reduce transmission rate.

Through our evaluation we find that a key requirement to make this approach feasible is to employ hardware packet pacing (a feature increasingly available in NICs) to smooth the transmission rate that results from widespread network features such as Large Send Offloading (LSO) and Interrupt Coalescing. We introduce innovative methods for estimating the congestion-friendly transmission rate of the pacer and for adaptively detecting the flows which require pacing. Without pacing, phantom queues would be fooled into regularly marking congestion based on spurious signals causing degradation in throughput, just as spikes in queuing caused by such bursting would hurt latency.

Taken together, we find that these techniques can reduce both average and 99th percentile packet latency by more than a factor of 10 compared to DCTCP and a factor of 40 compared to TCP. For example, in one configuration, the average latency drops from 78μs for DCTCP (329μs for TCP) to 7μs and the 99th percentile drops from 556μs for DCTCP (3961μs for TCP) to 48μs, with a configurable reduction in bandwidth for throughput-oriented applications. A factor of 10 reduction in latency has the potential to substantially increase the amount of work applications such as web search perform—e.g., process 10 times more data with predictable completion times—for end-user requests, though we leave such exploration of end-application benefits for future work.

2 Challenges and design

The goal of the HULL architecture is to *simultaneously deliver near baseline fabric latency and high throughput*. In this section we discuss the challenges involved in

[1]For example, queuing delays in a production cluster for a large-scale web application have been reported to range from $\sim$350μs at the median to over 14ms at the 99th percentile (see Figure 9 in [3]).

achieving this goal. These challenges pertain to correctly *detecting, signaling* and *reacting* to impending congestion. We show how these challenges guide our design decisions in HULL and motivate its three main components: *Phantom queues, DCTCP congestion control, and packet pacing*.

2.1 Phantom queues: Detecting and signaling congestion

The traditional congestion signal in TCP is the drop of packets. TCP increases its congestion window (and transmission rate) until available buffers overflow and packets are dropped. As previously discussed, given the low inherent propagation and switching times in the data center, this tail-drop behavior incurs an unacceptably large queuing latency.

Active queue management (AQM) schemes [18, 24, 6] aim to proactively signal congestion before buffers overflow. Most of these mechanisms try to regulate the queue around some target occupancy. While these methods can be quite effective in reducing queuing latency, they cannot eliminate it altogether. This is because they must observe a non-zero queue to begin signaling congestion, and sources react to these congestion signals after one RTT of lag, during which time the queue would have built up even further.

This leads to the following observation: Achieving predictable and low fabric latency essentially requires congestion signaling *before* any queueing occurs. That is, achieving the lowest level of queueing latency imposes a fundamental tradeoff with bandwidth—creating a 'bandwidth headroom'. Our experiments (§6) show that bandwidth headroom dramatically reduces average and tail queuing latencies. In particular, the reductions at the high percentiles are significant compared to queue-based AQM schemes.

We propose the Phantom Queue (PQ) as a mechanism for creating bandwidth headroom. A phantom queue is a simulated queue, associated with each switch egress port, that sets ECN [40] marks based on link utilization rather than queue occupancy. The PQ simulates queue buildup for a virtual egress link of a configurable speed, slower than the actual physical link (e.g., running at $\gamma = 95\%$ of the line rate). *The PQ is not really a queue since it does not store packets.* It is simply a counter that is updated while packets exit the link at line rate to determine the queuing that would have been present on the slower virtual link. It then marks ECN for packets that pass through it when the counter (simulated queue) is above a fixed threshold.

The PQ explicitly attempts to set aggregate transmission rates for congestion-controlled flows to be strictly less than the physical link capacity, thereby keeping switch buffers largely unoccupied. This bandwidth headroom allows latency sensitive flows to fly through the network at baseline transmission plus propagation rates.

Remark 1. The idea of using a simulated queue for signaling congestion has been used in Virtual Queue (VQ) AQM schemes [21, 31]. A key distinction is that while a VQ is typically placed in parallel to a real queue in the switch, we propose placing the PQ in *series* with the switch egress port. This change has an important consequence: the PQ can operate independently of the internal architecture of the switch (output-queued, shared memory, combined input-output queued) and its buffer management policies, and, therefore, work with *any* switch. In fact, we implement the PQ external to physical switches as a hardware 'bump on the wire' prototyped on the NetFPGA [33] platform (see §5.1). In general, of course, VQs and PQs can and have been [34] integrated into switching hardware.

2.2 DCTCP: Adaptive reaction to ECN

Standard TCP reacts to ECN marks by cutting the congestion window in half. Without adequate buffering to keep the bottleneck link busy, this conservative back off can result in a severe loss of throughput. For instance, with zero buffering, TCP's rate fluctuates between 50% and 100% of link capacity, achieving an average throughput of only 75% [51]. Therefore, since the PQ aggressively marks packets to keep the buffer occupancy at zero, TCP's back off can be especially detrimental.

To mitigate this problem, we use DCTCP [3], a recent proposal to enhance TCP's reaction to ECN marks. DCTCP employs a fixed marking threshold at the switch queue and attempts to extract information regarding the *extent* of network congestion from the sequence of congestion signals in the ACK train from the receiver. A DCTCP source calculates the fraction of packets containing ECN marks within a given window of ACKs, and reduces its window size in proportion to the fraction of marked packets. Essentially, congestion signals need not result in multiple flows simultaneously backing off drastically and losing throughput. Instead (and ideally), senders can adjust transmission rates to maintain a base, low level of queueing near the marking threshold. In theory, DCTCP can maintain more than 94% throughput even with zero queueing [4]. We refer to [3] for a full description of the algorithm.

2.3 Packet pacing

Bursty transmission occurs for multiple reasons, ranging from TCP artifacts like ACK compression and slow start [27, 56], to various offload features in NICs like Interrupt Coalescing and Large Send Offloading (LSO) designed to reduce CPU utilization [8]. With LSO for instance, hosts transfer large buffers of data to the NIC, leaving specialized hardware to segment the buffer into

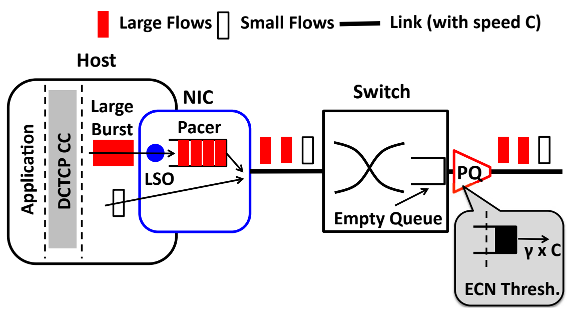


Figure 1: The HULL architecture consists of Phantom queues at switch egress ports and DCTCP congestion control and packet pacers at end-hosts.

individual packets, which then burst out at line rate. As our experiments show (§4.1), Interrupt Coalescing and LSO are required to maintain acceptable CPU overhead at 10Gbps speeds.

Traffic bursts cause temporary increases in queue occupancies. This may not be a big problem if enough buffer space is available to avoid packet drops and if variable latency is not a concern. However, since the PQ aggressively attempts to keep buffers empty (to prevent variable latency), such bursts trigger spurious congestion signals, leading to reduced throughput.

A natural technique for combating the negative effects of bursty transmission sources on network queueing is packet pacing. While earlier work (§7) introduce pacing at various points in the protocol stack, we find that, to be effective, pacing must take place in hardware after the last source of bursty transmission: NIC-based LSO. Ideally, a simple additional mechanism at the NIC itself would pace data transmission. The pacer would likely be implemented as a simple leaky bucket with a configurable exit rate. We describe a hardware design and implementation for pacing in §4.2.

It is paradoxical that the pacer must queue packets at the edge (end-hosts) so that queueing inside the network is reduced. Such edge queueing can actually increase end-to-end latency, offsetting any benefits of reduced in-network queueing. We resolve this paradox by noting that only packets that *belong to large flows* and hence are not sensitive to per-packet delivery times should be paced. Small latency-sensitive flows should not be paced, allowing them to exploit the lowest available fabric latency. We employ a simple adaptive end-host mechanism to determine whether a flow should be subject to pacing. This is inspired by classic work on the UNIX multi-level feedback queue that attempts to classify interactive versus bulk jobs in the operating system [46]. Newly created flows are classified as latency sensitive and initially not subjected to pacing. However, once a flow sees a sufficient number of ECN marks, it is classified as throughput-oriented and paced.

	Throughput	Mean Latency	99th Prctile Latency
TCP	982Mbps	1100.6μs	4308.8μs
DCTCP-30K	975Mbps	153.9μs	305.8μs

Table 1: Baseline throughput and latency for two long-lived flows with TCP and DCTCP-30K (30KB marking threshold).

2.4 The HULL Architecture

The complete High-bandwidth Ultra Low Latency (HULL) architecture is shown in Figure 1. Large flows at the host stack, which runs DCTCP congestion control, send large bursts to the NIC for segmentation via LSO. The Pacer captures the packets of the large flows after segmentation, and spaces them out at the correct transmission rate. The PQ uses ECN marking based on a simulated queue to create bandwidth headroom, limiting the link utilization to some factor, $\gamma < 1$, of the line rate. This ensures that switch queues run (nearly) empty, which enables low latency for small flows.

3 Bandwidth Headroom

This section explores the consequences of creating bandwidth headroom. We illustrate the role of the congestion control protocol in determining the amount of bandwidth headroom by comparing TCP and DCTCP. We then discuss how bandwidth headroom impacts the completion time of large flows.

3.1 Importance of stable rate control

All congestion control algorithms cause fluctuations in rate as they probe for bandwidth and react to delayed congestion signals from the network. The degree of these fluctuations is usually termed ‘stability’ and is an important property of the congestion control feedback system [47, 24]. Typically, some amount of buffering is required to absorb rate variations and avoid throughput loss. Essentially, buffering keeps the bottleneck link busy while sources that have cut their sending rates recover. This is especially problematic in low statistical multiplexing environments, where only a few high speed flows must sustain throughput [5].

Therefore, special care must be taken with the congestion control algorithm if we aim to reduce buffer occupancies to zero. We illustrate this using a simple experiment. We connect three servers to a single switch and initiate two long-lived flows from two of the servers to the third (details regarding our experimental setup can be found in §5). We measure the aggregate throughput and the latency due to queueing at the switch. As a baseline reference, the throughput and latency for standard TCP (with tail-drop), and DCTCP, with the recommended marking threshold of 30KB [3], are given in Table 1. As expected, DCTCP shows an order of magnitude improvement in latency over TCP, because it reacts to queue buildup beyond the marking threshold.

We conduct a series of experiments where we sweep the drain rate of a PQ attached to the congested port. The marking threshold at the PQ is set to 6KB and we also enable our hardware pacing module. The results are shown in Figure 2. Compared to the baseline, a significant latency reduction occurs for both TCP-ECN (TCP with ECN enabled) and DCTCP, when bandwidth headroom is created by the PQ. Also, for both schemes, the throughput is lower than intended by the PQ drain rate. This is because of the rate variations imposed by the congestion control dynamics. However, *TCP-ECN loses considerably more throughput than DCTCP at all PQ drain rates*. The gap between TCP-ECN's throughput and the PQ drain rate is ~17–26% of the line rate, while it is ~6–8% for DCTCP, matching theory quite well [4].

3.2 Slowdown due to bandwidth headroom

Bandwidth headroom created by the PQ will inevitably slow down the large flows, which are bandwidth-intensive. An important question is: *How badly will the large flows be affected?*

We answer this question using a simple queuing analysis of the 'slowdown', defined as the ratio of the completion times of a flow with and without the PQ. We find that, somewhat counter-intuitively, the slowdown is not simply determined by the amount of bandwidth sacrificed; it also depends on the traffic load.

Consider the well-known model of a M/G/1-Processor Sharing queue for TCP bandwidth sharing [20, 42]. Flows arrive according to a Poisson process of some rate, λ, and have sizes drawn from a general distribution, S. The flows share a link of capacity C in a fair manner; i.e., if there are n flows in the system, each gets a bandwidth of C/n. We assume the total load $\rho \triangleq \lambda \mathbb{E}(S)/C < 1$, so that the system is stable. A standard result for the M/G/1-PS queue states that in this setting, the average completion time for a flow of size x is given by:

$$FCT_{100\%} = \frac{x}{C(1-\rho)}, \tag{1}$$

where the '100%' indicates that this is the FCT without bandwidth headroom. Now, suppose we only allow the flows to use γC of the capacity. Noting that the load on this slower link is $\tilde{\rho} = \rho/\gamma$, and invoking (1) again, we find that the average completion time is:

$$FCT_{\gamma} = \frac{x}{\gamma C(1-\rho/\gamma)} = \frac{x}{C(\gamma-\rho)}. \tag{2}$$

Hence, dividing (2) by (1), the slowdown caused by the bandwidth headroom is given by:

$$SD \triangleq \frac{FCT_{\gamma}}{FCT_{100\%}} = \frac{1-\rho}{\gamma-\rho}. \tag{3}$$

The interesting fact is that *the slowdown gets worse as the load increases*. This is because giving bandwidth

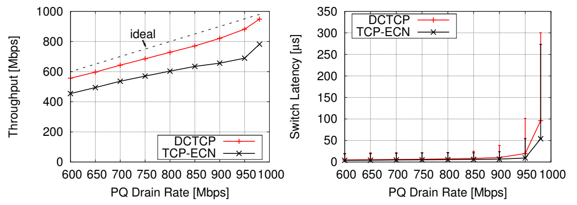


Figure 2: Throughput (left) and average switch latency (right) for TCP-ECN and DCTCP with a PQ, as drain rate varies. The vertical bars in the right plot indicate the 99th percentile.

away also increases the effective load ($\hat{\rho} > \rho$). For example, using (3), the slowdown with 20% bandwidth headroom ($\gamma = 0.8$), at load $\rho = 0.2, 0.4, 0.6$ will be 1.33, 1.5, 2, (equivalently: 33%, 50%, 100%) respectively.

Our experiments in §6.2 confirm the validity of this model (see Figure 10). This highlights the importance of not giving away too much bandwidth. Fortunately, as we show, *even a small amount of bandwidth headroom (e.g., 10%) provides a dramatic reduction in latency*.

Remark 2. The M/G/1-PS model provides a good approximation for large flows for which TCP has time to converge to the fair bandwidth allocation [20]. It is not, however, a good model for small flows as it does not capture latency. In fact, since the completion time for small flows is mainly determined by the latency, they are not adversely affected by bandwidth headroom (§6).

4 Pacing

4.1 The need for pacing

Modern NICs implement various offload mechanisms to reduce CPU overhead for network communication. These offloads typically result in highly bursty traffic [8]. For example, Interrupt Coalescing is a standard feature which allows the NIC to delay interrupting the CPU and wait for large batches of packets to be processed in one SoftIrq. This disrupts the normal TCP ACK-clocking and leads to many MTUs worth of data being released by TCP in a burst. A further optimization, Large Send Offloading (LSO), allows TCP to send large buffers (currently up to 64KB), delegating the segmentation into MTU-sized packets to the NIC. These packets then burst out of the NIC at line rate.

As later experiments show, this burstiness can be detrimental to network performance. However, *using hardware offloading to reduce the CPU overhead of the network stack is unavoidable as link speeds increase to 10Gbps and beyond.*

We illustrate the need for pacing using a simple experiment. We directly connect two servers with 10Gbps NICs (see §5.2 for testbed details), and enable LSO and Interrupt Coalescing with a MTU of 1500 bytes. We generate a single TCP flow between the two servers, and cap the window size of the flow such that the throughput is ~1Gbps on average. Figure 3 shows the data and

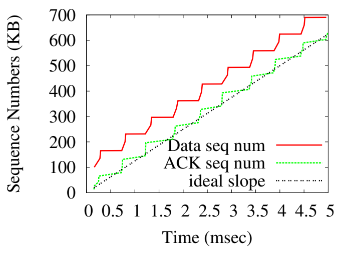


Figure 3: Burstiness with 10Gbps NICs.

Intr. Coalescing	CPU Util. (%)	Thrput (Gbps)	Ack Ratio (KB)
adaptive	37.2	9.5	41.3
rx-frames=128	30.7	9.5	64.0
rx-frames=32	53.2	9.5	16.5
rx-frames=8	75	9.5	12.2
rx-frames=2	98.7	9.3	11.4
rx-frames=0	99	7.7	67.4

Table 2: The effect of Interrupt Coalescing. Note that 'adaptive' is the default setting for Interrupt Coalescing.

ACK sequence numbers within a 5ms window (time and sequence numbers are relative to the origin) compared to the 'ideal slope' for perfectly paced transmission at 1Gbps. The sequence numbers show a step-like behavior that demonstrates the extent of burstiness. Each step, occurring roughly every 0.5ms, corresponds to a back-to-back burst of data packets totaling 65KB. Analyzing the packet trace using `tcpdump` [49], we find that the bursty behavior reflects the batching of ACKs at the receiver: Every 0.5ms, 6 ACKs acknowledging 65KB in total are received within a 24–50μs interval. Whereas, ideally, for a flow at 1Gbps, the ACKs for 65KB should be evenly spread over 520μs.

The batching results from Interrupt Coalescing at the receiver NIC. To study this further, we repeat the experiment with no cap on the window size and different levels of Interrupt Coalescing. We control the extent of Interrupt Coalescing by varying the value of `rx-frames`, a NIC parameter that controls the number of frames between interrupts. Table 2 summarizes the results. We observe a tradeoff between the CPU overhead at the receiver and the average ACK ratio (the number of data bytes acknowledged by one ACK), which is a good proxy for burstiness. Setting `rx-frames` at or below 8 heavily burdens the receiver CPU, but improves the ACK ratio. With Interrupt Coalescing disabled (`rx-frames` = 0), the receiver CPU is saturated and cannot keep up with the load. This further increases the ACK ratio and also causes a 1.8Gbps loss in throughput.

Remark 3. Enabling LSO is also necessary to achieve the 10Gbps line rate. Without LSO, the sender's CPU is saturated and there is close to 3Gbps loss of throughput.

4.2 Hardware Pacer module

The Pacer module inserts suitable spacing between the packets of flows transmitted by the server. We envision the Pacer module operating at the NIC. Pacing in hardware has a number of advantages over software pacing. A hardware pacer can easily support the sub-microsecond scheduling granularity required to pace at 10Gbps rates and beyond. Moreover, unlike pacing in the host stack that typically requires disabling segmentation offload, a hardware module in the NIC is oblivious to server LSO settings since it operates on the outgoing packet stream *after* segmentation takes place.

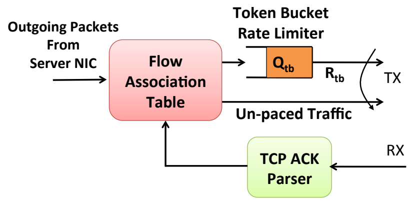


Figure 4: Block diagram of Pacer module.

Figure 4 shows the block diagram of the Pacer module. The Flow Association Table is consulted to check whether an outgoing packet requires pacing (see below). If so, the packet is placed into a token bucket rate limiter with a configurable transmission rate. Otherwise, it bypasses the token bucket and is sent immediately.

The key challenges to pacing, especially in a hardware module, are: (i) determining the appropriate pacing rate, and (ii) deciding the flows that require pacing.

Dynamic pacing rate estimation. The NIC is unaware of the actual sending rate ($Cwnd/RTT$) of TCP sources. Therefore, we use a simple algorithm to estimate the congestion-friendly transmission rate. We assume that over a sufficiently large measurement interval (e.g., a few RTTs) each host's aggregate transmission rate will match the rate imposed by higher-level congestion control protocols such as TCP (or DCTCP). The pacer dynamically measures this rate and appropriately matches the rate of the token bucket. More precisely, every T_r seconds, the Pacer counts the number of bytes it receives from the host, denoted by M_r. It then modifies the rate of the token bucket according to:

$$R_{tb} \leftarrow (1-\eta) \times R_{tb} + \eta \times \frac{M_r}{T_r} + \beta \times Q_{tb}. \qquad (4)$$

The parameters η and β are positive constants and Q_{tb} is the current backlog of the token bucket in bytes. R_{tb} is in bytes per second.

Equation (4) is a first order low-pass filter on the rate samples M_r/T_r. The term $\beta \times Q_{tb}$ is necessary to prevent the Pacer backlog from becoming too large.[2] This is crucial to avoid a large buffer for the token bucket, which adds to the cost of the Pacer, and may also induce significant latency to the paced flows (we explore the latency of the Pacer further in §4.4).

[2]In fact, if the aggregate rate of paced flows is fixed at $R^* = M_r/T_r$, the only fixed point of equation (4) is $R_{tb} = R^*$, and $Q_{tb} = 0$.

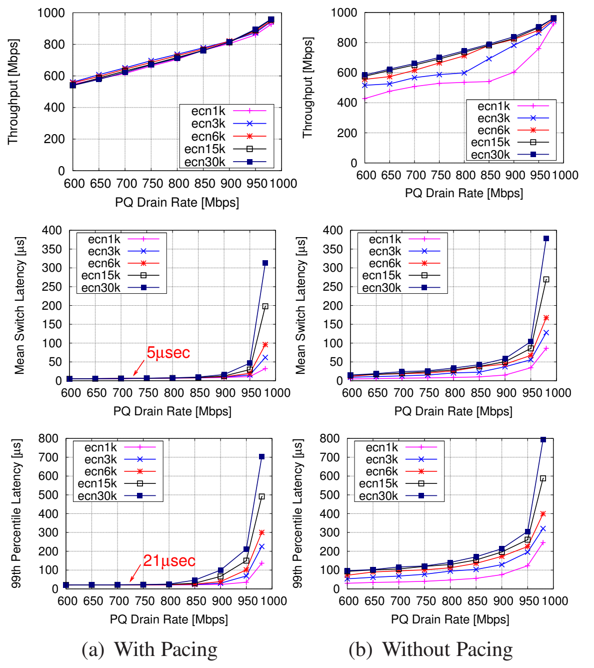


Figure 5: Throughput and switch latency as PQ drain rate varies, with and without pacing.

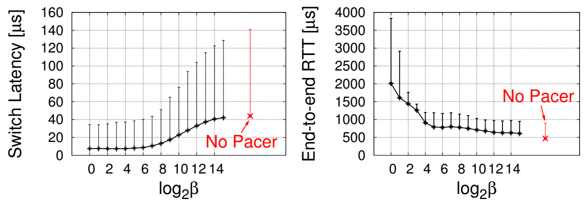


Figure 6: The average switch latency and end-to-end RTT (measured by ping) for paced flows, as β in Eq. (4) is varied.

Which flows need pacing? As previously discussed, only packets belonging to large flows (that are not latency-sensitive) should be paced. Moreover, only those flows that *are causing congestion* require pacing. We employ a simple adaptive mechanism to automatically detect such flows. Newly created flows are initially not paced. For each ACK with the ECN-Echo bit set, the corresponding flow is associated with the pacer with some probability, p_a (e.g., 1/8 in our implementation). This probabilistic sampling ensures that small flows are unlikely to be paced. The association times out after some time, T_i, so that idle flows will eventually be reclassified as latency sensitive.

Remark 4. We have described the pacer module with a single token bucket rate-limiter for simplicity. However, the same design can be used with multiple rate-limiters, allowing more accuracy when pacing multiple flows.

4.3 Effectiveness of the Pacer

We demonstrate the effectiveness of pacing by running an experiment with 2 long-lived DCTCP flows (similar to §3.1), with and without pacing. As before, we sweep the drain rate of the PQ. We also vary the marking threshold at the PQ from 1KB to 30KB. The results are shown in Figure 5. We observe that pacing improves both throughput and latency. The throughput varies nearly linearly with the PQ drain rate when the Pacer is enabled. Without pacing, however, we observe reduced throughput with low marking thresholds. This is because of spurious congestion signals caused by bursty traffic. Also, with pacing, the average and 99th percentile latency plummet with bandwidth headroom, quickly reaching their floor values of 5μs and 21μs respectively. In contrast, the latency decreases much more gradually without pacing, particularly at the 99th percentile.

4.4 The tradeoff between Pacer delay and effectiveness

Pacing, *by definition*, implies delaying the transmission of packets. We find that there is a tradeoff between the delay at the Pacer and how effectively it can pace. This tradeoff is controlled by the parameter β in Equation (4). Higher values of β cause a more aggressive increase in the transmission rate to keep the token bucket backlog, Q_{tb}, small. However, this also means that the Pacer creates more bursty output when a burst of traffic hits the token bucket; basically, the Pacer does 'less pacing'.

The following experiment shows the tradeoff. We start two long-lived DCTCP flows transmiting to a single receiver. The Pacer is enabled and we sweep β over the range $\beta = 2^0$ to $\beta = 2^{14}$ (the rest of the parameters are set as in Table 3). The PQ on the receiver link is configured to drain at 950Mbps and has a marking threshold of 1KB. We measure both the latency across the switch and the end-to-end RTT *for the flows being paced*, which is measured using ping from the senders to the receiver.

Figure 6 shows the results. The Pacer is more effective in reducing switch latency with smaller value of β, but also induces more delay. We observe a sharp increase in Pacer delay for values of β smaller than 2^4 without much gain in switch latency, suggesting the sweet spot for β. Nonetheless, the Pacer does add a few hundreds of microseconds of delay to the paced flows. This underscores the importance of selectively choosing the flows to pace. Only large flows which are throughput-oriented and are not impacted by the increase in delay should be paced. In fact, the throughput (not shown due to lack of space) is also higher with better pacing: Decreasing from ~870Mbps at $\beta = 2^0$ to ~770Mbps at $\beta = 2^{14}$ (and slightly lower without pacing).

5 Experimental Setup

5.1 Implementation

We use the NetFPGA [33] platform to implement the Pacer and PQ modules. NetFPGA is a PCI card with

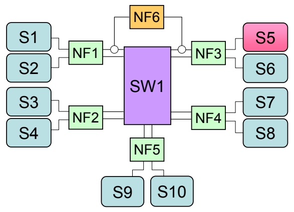


Figure 7: Testbed topology.

four Gigabit Ethernet ports and a Xilinx Virtex-II Pro 125MHz FPGA that has 4MB of SRAM. Each NetFPGA supports two Pacers and two PQs. The complete implementation consumes 160 block RAMs (out of 232, or 68% of the total NetFPGA capacity), and occupies 20,364 slices (86% of the total NetFPGA capacity).

The Pacer module has a single token bucket rate-limiter with a 128KB FIFO queue. The Pacer's transmission rate is controlled as described in §4.2 at a granularity of 0.5Mbps. Tokens of variable size (1-2000 bytes, proportional to the Pacer rate) are added every 16μs and the maximum allowed outstanding tokens (bucket depth) is 3KB. The Flow Association Table can hold 64 entries, identifying flows that require pacing. The Pacer enqueues the packets of these flows in the rate-limiter and forwards the rest in pass-through mode.

The PQ module implements the virtual queue length counter. It is incremented upon receiving a packet and decremented according to the PQ drain rate every 800ns. If the counter exceeds the configured marking threshold, the ECN/CE (Congestion Experienced) bit in the IP header of the incoming packet is set and the checksum value is recalculated. The PQ is completely pass-through and does not queue packets.

We have also introduced modifications to the TCP stack in Linux 2.6.26 for DCTCP, following the algorithm in [3]. Our code is available online at [13].

5.2 Testbed

Our testbed consists of 10 servers and 6 NetFPGAs connected to a Broadcom Triumph2 switch as shown in Figure 7. The Triumph2 is an ECN-capable switch with 48 nonblocking 1Gbps ports and 4MB of buffer memory shared across all ports. Each server has 4-core Intel Xeon E5620 2.4GHz CPUs with Hyper-Threading and at least 16GB of RAM. The servers use Intel's 82574L 1GbE Ethernet Controller. Two of the servers, S9 and S10, also have Mellanox ConnectX-2 ENt 10Gbase-T NICs, which were used for the 10Gbps experiments in §4.1.

Each of the NetFPGAs NF1-NF5 implements two Pacers and two PQs: One for each of the two servers and the two switch ports connected to it. All server-to-switch traffic goes through the Pacer module and all switch-to-server traffic goes through the PQ module.

For the majority of the experiments in this paper, we use machine S5 as the receiver, and (a subset of) the rest of the machines as senders which cause congestion at the switch port connected to S5 (via NF3).

Measuring Switch Latency. We have also developed a Latency Measurement Module (LMM) in NetFPGA for sub-microsecond resolution measurement of the latency across the congested switch port. The LMM (NF6 in Figure 7) works as follows: Server S1 generates a 1500 byte ping packet to S5 every 1ms.[3] The ping packets are intercepted and timestamped by the LMM before entering the switch. As a ping packet leaves the switch, it is again intercepted and the previous time-stamp is extracted and subtracted from the current time to calculate the latency.

6 Results

This section presents our experimental and simulation results evaluating HULL. We use micro-benchmarks to compare the latency and throughput performance of HULL with various schemes including TCP with drop-tail, default DCTCP, DCTCP with reduced marking threshold, and TCP with an ideal two-priority QoS scheme (TCP-QoS), where small (latency-sensitive) flows are given strict priority over large flows. We also check scalability of HULL using large-scale ns-2 [36] simulations. We briefly summarize our main findings:
(i) In micro benchmarks with both static and dynamic traffic, we find that HULL significantly reduces average and tail latencies compared to TCP and DCTCP. For example, with dynamic traffic (§6.2) HULL provides a more than 40x reduction in average latency compared to TCP (more than 10x compared to DCTCP), with bigger reductions at the high percentiles. Compared to an optimized DCTCP with low marking threshold and pacing, HULL achieves a 46–58% lower average latency, and a 69–78% lower 99th percentile latency. The bandwidth traded for this latency reduction increases the completion-time of large flows by 17–55%, depending on the load, in good agreement with the theoretical prediction in §3.2.
(ii) HULL achieves comparable latency to TCP-QoS with two priorities, but lower throughput since QoS does not leave bandwidth headroom. Also, unlike TCP-QoS which loses a lot of throughput if buffers are shallow (more than 58% in one experiment), HULL is much less sensitive to the size of switch buffers, as it (mostly) keeps them unoccupied.
(iii) Our large-scale ns-2 simulations confirm that HULL scales to large multi-switch topologies.

Parameter choices. Table 3 gives the baseline parameters used in the testbed experiments (the ns-2 parameters are given in §6.3). The parameters are determined experimentally. Due to space constraints, we omit the details

[3]Note that this adds 12Mbps (1.2%) of throughput overhead.

Phantom Queue	Drain Rate = 950Mbps, Marking Thresh. = 1KB
Pacer	$T_r = 64\mu$s, $\eta = 0.125$, $\beta = 16$, $p_a = 0.125$, $T_i = 10ms$

Table 3: Baseline parameter settings in experiments.

and summarize our main findings.

The PQ parameters are chosen based on experiments with long-lived flows, like that in Figure 5. As can be seen in this figure, the PQ with 950Mbps drain rate and 1KB marking threshold (with pacing) achieves almost the latency floor. An interesting fact is that smaller marking thresholds are required to maintain low latency as the PQ drain rate (γC) increases. This can be seen most visibly in Figure 5(a) for the 99th percentile latency. The reason is that since the input rate into the PQ is limited to the line rate (because it's in series), it takes longer for it to build up as the drain rate increases. Therefore, the marking threshold must also be reduced with increasing drain rate to ensure that the PQ reacts to congestion quickly.

Regarding the Pacer parameters, we find that the speed of the Pacer rate adaptation—determined by T_r/η—needs to be on the order of a few RTTs. This ensures that the aggregate host transmission rate is tracked closely by the Pacer and provides a good estimate of the rate imposed by the higher-layer DCTCP congestion control. The parameter β is chosen as described in §4.4. The parameters p_a and T_i are chosen so that small flows (e.g., smaller than 10KB) are unlikely to be paced. Overall, we do not find the Pacer to be sensitive to these parameters.

6.1 Static Flow Experiments

We begin with an experiment that evaluates throughput and latency in the presence of long-lived flows. We call this the *static* setting since the number of flows is constant during the experiment and the flows always have data to send. Each flow is from a different server sending to the receiver S5 (see Figure 7). We sweep the number of flows from 2 to 8. (Note that at least 2 servers must send concurrently to cause congestion at the switch.)

Schemes. We compare four schemes: (i) standard TCP (with drop-tail), (ii) DCTCP with 30KB marking threshold, (iii) DCTCP with 6KB marking threshold and pacing enabled, and (iv) DCTCP with a PQ (950Mbps with 1KB marking threshold). For schemes (ii) and (iii), ECN marking is enabled at the switch and is based on the physical queue occupancy, while for (iv), marking is only done by the PQ.

Note: The recommended marking threshold for DCTCP at 1Gbps is 30KB [3]. We also lower the marking threshold to 6KB to evaluate how much latency can be improved with pure queue-based congestion signaling. Experiments show that with this low marking threshold, pacing is required to avoid excessive loss in throughput (§6.2.1). Reducing the marking threshold below 6KB

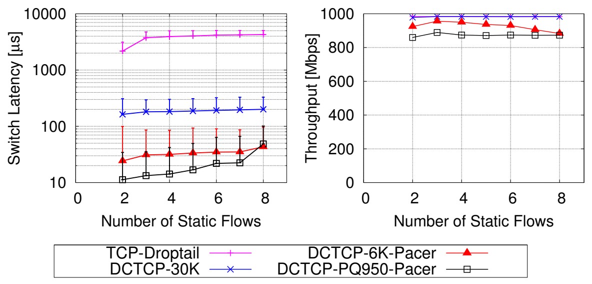


Figure 8: Switch latency (left) and throughput (right) as the number of long-lived flows varies. The latency plot uses a logarithmic scale and the vertical bars indicate the 99th percentile.

severely degrades throughput, even with pacing.

Analysis. The results are shown in Figure 8. We observe more than an order of magnitude (about 20x) reduction in average latency with DCTCP compared to TCP. Reducing the marking threshold to 6KB gives a further 6x reduction, bringing the average latency down from $\sim$200μs to $\sim$30μs. When there are a few static flows (e.g., less than 4), the PQ reduces the average latency by another factor 2–3 compared to DCTCP with 6KB marking threshold. Moreover, it also significantly lowers the jitter, achieving a 99th percentile of $\sim$30μs compared to $\sim$100μs for DCTCP-6K-Pacer, and more than 300μs for standard DCTCP. The PQ's lower latency is because of the bandwidth headroom it creates: The throughput for the PQ is about 870Mbps. The 8% loss compared to the PQ's 950Mbps drain rate is due to the rate fluctuations of DCTCP, as explained in §3.1.

Behavior with increasing flows. Figure 8 shows that bandwidth headroom becomes gradually less effective with increasing the number of flows. This is because of the way TCP (and DCTCP) sources increase their window size to probe for additional bandwidth. As is well-known, during Congestion Avoidance, a TCP source increases its window size by one packet every round-trip time. This is equivalent to an increase in sending rate of $1/RTT$ (in pkts/sec) each round-trip-time. Now, with N flows all increasing their rates at this slope, more bandwidth headroom is required to prevent the aggregate rate from exceeding the link capacity and causing queuing. More precisely, because of the one RTT of delay in receiving ECN marks from the PQ, the sources' aggregate rate overshoots the PQ's target drain rate by N/RTT (in pkts/sec). Hence, we require:

$$(1-\gamma)C > \frac{N}{RTT} \Longrightarrow 1-\gamma > \frac{N}{C \times RTT}, \qquad (5)$$

where $C \times RTT$ is the bandwidth-delay product in units of packets. Equation (5) indicates that *the bandwidth headroom required (as a percentage of capacity) to prevent queuing increases with more flows and decreases with larger bandwidth-delay product.*

		Switch Latency (μs)			1KB FCT (μs)			10MB FCT (ms)		
		Avg	**90th**	**99th**	**Avg**	**90th**	**99th**	**Avg**	**90th**	**99th**
20% Load	**TCP-DropTail**	111.5	450.3	1224.8	1047	1136	10533	110.2	162.3	349.6
	DCTCP-30K	38.4	176.4	295.2	475	638	2838	**106.8**	**155.2**	**301.7**
	DCTCP-6K-Pacer	6.6	13.0	59.7	389	531	888	111.8	168.5	320.0
	DCTCP-PQ950-Pacer	**2.8**	**7.6**	**18.6**	**380**	**529**	**756**	125.4	188.3	359.9
40% Load	**TCP-DropTail**	329.3	892.7	3960.8	1537	3387	5475	**151.3**	**275.3**	575.0
	DCTCP-30K	78.3	225.0	556.0	495	720	1794	155.1	281.5	**503.3**
	DCTCP-6K-Pacer	15.1	35.5	213.4	403	560	933	168.7	309.3	567.5
	DCTCP-PQ950-Pacer	**7.0**	**13.5**	**48.2**	**382**	**536**	**808**	198.8	370.5	654.7
60% Load	**TCP-DropTail**	720.5	2796.1	4656.1	2103	4423	5425	**250.0**	**514.6**	1007.4
	DCTCP-30K	119.1	247.2	604.9	511	740	1268	267.6	538.4	**907.3**
	DCTCP-6K-Pacer	24.8	52.9	311.7	403	563	923	320.9	632.6	1245.6
	DCTCP-PQ950-Pacer	**13.5**	**29.3**	**99.2**	**386**	**530**	**782**	389.4	801.3	1309.9

Table 4: Baseline dynamic flow experiment. The average, 90th percentile, and 99th percentile switch latency and flow completion times are shown. The results are the average of 10 trials. In each case, the best scheme is shown in red.

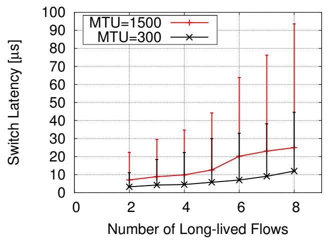


Figure 9: The impact of increasing number of flows on switch latency with PQ, for MTU = 1500 bytes and MTU = 300 bytes.

An important consequence of Equation (5) is that for a fixed RTT, less bandwidth headroom is needed as the link capacity increases (e.g., from 1Gbps to 10Gbps). We demonstrate this using a simple experiment in Figure 9. In the absence of an experimental setup faster than 1Gbps, we emulate what happens at higher link speeds by decreasing the MTU (packet size) to 300 bytes. Since the default MTU is 1500 bytes, this increases the bandwidth-delay product by a factor of 5 in units of packets, effectively emulating a 5Gbps link. As can be seen in the figure, the latency with the PQ is much lower with the smaller packet size at all number of flows. This confirms that the *sensitivity to the number of flows decreases with increasing link speed. Essentially, the same amount of bandwidth headroom is more effective for faster links.*

Remark 5. We found that when the MTU is reduced to 300 bytes, the receiver NIC cannot keep up with the higher packets/sec and starts dropping packets. To avoid this artifact, we had to reduce the PQ drain rate to 800Mbps for the tests in Figure 9.

6.2 Dynamic Flow Experiments

In this section we present the results of a micro-benchmark which creates a *dynamic* workload. We develop a simple client/server application to generate traffic based on patterns seen in storage systems like memcached [54]. The client application, running on server S5 (Figure 7) opens 16 permanent TCP connections with each of the other 9 servers. During the test, the client repeatedly chooses a random connection among the pool of connections and makes a request for a file on that connection. The server application responds with the requested file. The requests are generated as a Poisson process in an open loop fashion [43]; that is, new requests are triggered independently of prior outstanding requests. The request rate is chosen such that the average RX throughput at the client is at a desired level of load. For example, if the average file size is 100KB, and the desired load is 40% (400Mbps), the client makes 500 requests per second on average. We conduct experiments at low (20%), medium (40%), and high (60%) levels of load. During the experiments, we measure the switch latency (using the NetFPGA Latency Measurement Module), as well as the application level flow completion times (FCT).

6.2.1 Baseline

For the baseline experiment, we use a workload where 80% of all client requests are for a 1KB file and 20% are for a 10MB file. Of course, this is not meant to be a realistic workload. Rather, it allows a clear comparison of how different schemes impact the small (latency-sensitive) and large (bandwidth-sensitive) flows.

Note: The 1KB flows are just a single packet and can complete in one RTT. Such single packet flows are very common in data center networks; for example, measurements in a production data center of a large cloud service provider have revealed that more than 50% of flows are smaller than 1KB [22].

Table 4 gives the results for the same four schemes that were used in the static flow experiments (§6.1).

Analysis: Switch latency. The switch latency is very high with TCP compared to the other three schemes since it completely fills available buffers. With DCTCP, the latency is 3–6 times lower on average. Reducing the marking threshold to 6KB gives another factor of 5 reduction in average latency. However, some baseline level of queueing delay and significant *jitter* remains, with hundreds of microseconds of latency at the 99th percentile.

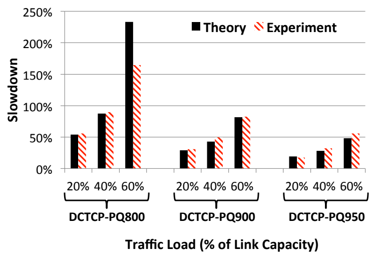


Figure 10: Slowdown for 10MB flows: Theory vs Experiment.

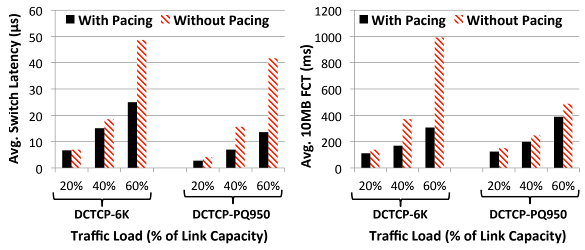


Figure 11: Average switch latency (left) and 10MB FCT (right), with and without pacing.

As explained in §2.1, this is because queue-based congestion signaling (even with pacing) is too late to react; by the time congestion is detected, a queue has already built up and is increasing. The lowest latency is achieved by the PQ which creates bandwidth headroom. Compared to DCTCP-6K-Pacer, the PQ reduces the average latency by 46–58% and the 99th percentile by 69–78%. The latency with the PQ is especially impressive considering just a single 1500 byte packet adds 12μs of queueing latency at 1Gbps.

Analysis: 1KB FCT & End-host latency. The 1KB FCT is rather high for all schemes. It is evident from the switch latency measurements that the high 1KB FCTs are due to the delays incurred by packets at the end-hosts (in the software stack, PCIe bus, and network adapters). The host-side latency is particularly large when the servers are actively transmitting/receiving data at high load. As a reference, the minimum 1KB FCT is about 160μs. Interestingly, the latency at the end-host (and the 1KB FCT) improves with more aggressive signaling of congestion in the network, especially, compared to TCP-DropTail. This suggests that the un-checked increase in window size with TCP-DropTail (and to a lesser extent with DCTCP-30K) causes queuing at both the network *and* the end-hosts. Essentially, flows with large windows deposit large chunks of data into NIC buffers, which adds delay for the small flows.

The main takeaway is that *bandwidth headroom significantly reduces the average and tail switch latency under load, even compared to optimized queue-based AQM with a low marking threshold and pacing. However, to take full advantage of this reduction, the latency of the software stack and network adapters must also improve.*

Analysis: Slowdown for 10MB flows. The bandwidth given away by the PQ increases the flow completion of the 10MB flows, which are throughput-limited. As predicted by the theoretical model in §3.2, the slowdown is worse at higher load. Compared to the lowest achieved value (shown in red in Table 4), with the PQ, the average 10MB FCT is 17% longer at 20% load, 31% longer at 40% load, and 55% longer at 60% load. Figure 10 compares the slowdown predicted by theory with that observed in experiments. The comparison includes the results for PQ with 950Mbps drain rate, given in Table 4, as well as PQ with 900Mbps and 800Mbps for which the detailed numbers are omitted in the interest of space.

The theoretical slowdown is computed using Equation (3). To invoke the equation, we use $\rho = 0.2, 0.4$, and 0.6 corresponding to the load. We also account for the additional throughput loss due to DCTCP rate fluctuations (§3.1), by subtracting 8% from the PQ drain rate to get γ. That is, we use $\gamma = 0.72, 0.82$, and 0.87 corresponding to the drain rates 800, 900, and 950Mbps. Overall, the theory and experiments match very well.

Pacing vs No Pacing. Figure 11 compares the average switch latency and 10MB FCT, with and without pacing. The comparison is shown for DCTCP with the marking threshold at the switch set to 6KB, and for DCTCP with the PQ draining at 950Mbps. In all cases, pacing lowers both the switch latency and the FCT of large flows, improving latency *and* throughput.

Remark 6. Most data center networks operate at loads less than 30% [9], so a load of 60% with Poisson/bursty traffic is highly unlikely—the performance degradation would be too severe. The results at 20% and 40% load are more indicative of actual performance.

6.2.2 Comparison with two-priority QoS scheme

Ethernet and IP provide multiple Quality of Service (QoS) priorities. One method for meeting the objective of ultra-low latency and high bandwidth is to use two priorities: an absolute priority for the flows which require very low latency and a lower priority for the bandwidth-intensive elastic flows. While this method has the potential to provide ideal performance, it may be impractical (and is not commonly deployed) because applications do not segregate latency-sensitive short flows and bandwidth-intensive large flows dynamically. Indeed, application developers do not typically consider priority classes for network transfers. It is more common to assign an entire application to a priority and use priorities to segregate *applications*.

Despite this, we now compare HULL with TCP using an ideal two-priority QoS scheme for benchmarking purposes. We repeat the baseline experiment from the previous section, but for QoS, we modify our application to classify the 1KB flows as 'high-priority' using the Type

		Switch Latency (μs)			1KB FCT (μs)			10MB FCT (ms)		
		Avg	**90th**	**99th**	**Avg**	**90th**	**99th**	**Avg**	**90th**	**99th**
Large Buffer	**TCP-QoS**	**6.4**	17.2	**20.2**	565	570	2308	**152.6**	**275.4**	**585.1**
	DCTCP-PQ950-Pacer	6.9	**13.5**	47.8	**381**	**538**	**810**	199.0	370.2	658.9
Small Buffer	**TCP-QoS**	5.3	15.5	**19.7**	**371**	**519**	**729**	362.3	811.9	1924.3
	DCTCP-PQ950-Pacer	**5.0**	**13.2**	35.1	378	521	759	**199.4**	**367.0**	**654.9**

Table 5: HULL vs QoS with two priorities. The switch buffer is configured to use either Dynamic buffer allocation (Large Buffer) or a fixed buffer of 10pkts = 15KB (Small Buffer). In all tests, the total load is 40%. In the case of QoS, the switch latency is that of the high priority queue. The results are the average of 10 trials.

of Service (TOS) field in the IP header. The switch uses a separate queue for these flows which is given strict priority over the other, 'best-effort', traffic.

Scenarios. We consider two settings for the switch buffer size: (i) Dynamic buffer allocation (the default settings in the switch), and (ii) a fixed buffer of 10 packets (15KB) per priority. Note that in the latter setting TCP-QoS gets 30KB in total, whereas HULL gets just 15KB since it always uses only one priority. The second setting is used to evaluate how switches with very shallow buffers impact performance, since dynamic buffer allocation allows a congested port to grab up to $\sim$700KB of the total 4MB of buffer in the switch).

Analysis: HULL vs QoS. Table 5 gives the results. We make three main observations:

(i) HULL and QoS achieve roughly the same average switch latency. HULL is slightly better at the 90th percentile, but worse at the 99th percentile.

(ii) When the switch buffer is large, TCP-QoS achieves a better FCT for the 10MB flows than HULL as it does not sacrifice any throughput. However, with small buffers, there is about a 2.4x increase in the FCT with TCP-QoS (equivalent to a 58% reduction in throughput). HULL achieves basically the same throughput in both cases because it does not need the buffers in the first place.

(iii) In the large buffer setting, the 1KB flows complete significantly faster with HULL—more than 33% faster on average and 65% faster at the 99th percentile. This is because the best-effort flows (which have large window sizes) interfere with high-priority flows at the end-hosts, similar to what was observed for TCP-DropTail in the baseline experiment (Table 4). This shows that all points of contention (including the end-hosts, PCIe bus, and NICs) must respect priorities for QoS to be effective.

Overall, this experiment shows that *HULL achieves nearly as low a latency as the ideal QoS scheme, but gets lower throughput. Also, unlike QoS, HULL can cope with switches with very shallow buffers because it avoids queue buildup altogether.*

6.3 Large-scale ns-2 Simulation

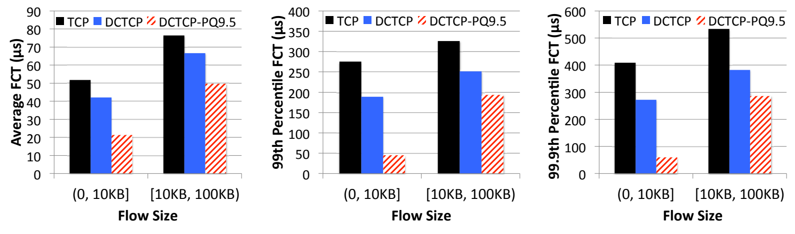
Due to the small size of our testbed, we cannot verify in hardware that HULL scales to the multi-switch topologies common in data centers. Therefore, we complement our hardware evaluation with large-scale ns-2 simulations targeting a multi-switch topology and workload.

Topology. We simulate a three-layered fat-tree topology based on scalable data center architectures recently proposed [2, 22]. The network consists of 56 8-port switches that connect 192 servers organized in 8 pods. There is a 3:1 over-subscription at the top-of-the-rack (TOR) level. The switches have 250 packets worth of buffering. All links are 10Gbps and have 200ns of delay, with 1μs of additional delay at the end-hosts. This, along with the fact that ns-2 simulates *store-and-forward* switches, implies that the end-to-end round-trip latency for a 1500 byte packet and a 40 byte ACK across the entire network (6 hops) is 11.8μs.

Routing. We use standard Equal-Cost Multi-Path (ECMP) [7] routing. Basically, ECMP hashes each flow to one of the shortest paths between the source and destination nodes. All packets of the flow take the same path. This avoids the case where packet reordering is misinterpreted as a sign of congestion by TCP (or DCTCP).

Workload. The workload is generated similarly to our dynamic flow experiments in hardware. We open permanent connections between each pair of servers. Flows arrive according to a Poisson process and are sent from a random source to a random destination server. The flow sizes are drawn from a Pareto distribution with shape parameter 1.05 and mean 100KB. This distribution creates a heavy-tailed workload where the majority of flows are small, but the majority of traffic is from large flows, as is commonly observed in real networks: 95% of the flows are less than 100KB and contribute a little over 25% of all data bytes; while 0.03% of the flows that are larger than 10MB contribute over 40% of all bytes.

Simulation settings. We compare standard TCP, DCTCP, and DCTCP with a PQ draining at 9.5Gbps (HULL). The Pacer is also enabled for HULL, with parameters: $T_r = 7.2\mu$s, $\eta = 0.125$, $\beta = 375$, $p_a = 0.125$, and $T_i = 1$ms. The changes to the parameters compared to the ones we used for the hardware Pacer (Table 3) are because of the difference in link speed (10Gbps vs 1Gbps) and the much lower RTT in the simulations. We set the flow arrival rate so the load at the server-to-TOR links is 15% (We have also run many simulations with other levels of load, with qualitatively similar results).



(a) Average (b) 99th Percentile (c) 99.9th Percentile

Figure 12: Average and high percentile FCT for small flows.

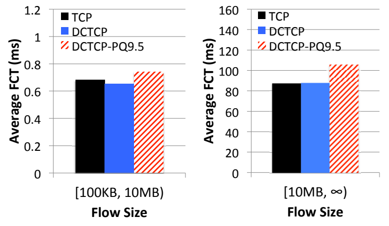


Figure 13: Average FCT for large flows.

All simulations last for at least 5 million flows.

Note: Because the topology has 3:1 over-subscription at the TOR, the load is approximately 3 times higher at the TOR-to-Agg and Agg-to-Core links. A precise calculation shows that the load is 43.8% at the TOR-to-Agg links, and 39.6% at the Agg-to-Core links.

Analysis: Small flows. Figure 12 shows the average, 99th percentile, and 99.9th percentile of the FCT for small flows. The statistics are shown separately for flows smaller than 10KB, and flows with size between 10KB and 100KB. We observe a significant improvement in the FCT of these flows with HULL; especially for flows smaller than 10KB, there is more than 50% reduction in the average FCT compared to DCTCP, and more than 80% reduction at the 99th and 99.9th percentiles.

It is important to note that the 20μs average FCT for flows smaller than 10KB achieved by HULL is near ideal given that the simulation uses store-and-forward switching. In fact, the average size for flows smaller than 10KB is 6.8KB. Because of store-and-forward, the FCT for a flow of this size is at least $\sim$16.2μs. This implies that with HULL, across the 5 switches end-to-end between source and destination, there are, *in total*, only 3 packets being queued on average (each adding 1.2μs of delay).

Analysis: Large flows. Figure 13 shows the average FCT for flows between 100KB and 10MB in size, and for those that are larger than 10MB. As expected, these flows are slower with HULL: up to 24% slower for flows larger than 10MB, which is approximately the slowdown predicted by theory (§3.2) at this fairly high level of load.

Overall, the ns-2 simulations confirm that bandwidth headroom created by HULL is effective in large multi-switch networks and can significantly reduce latency and jitter for small flows.

7 Related Work

AQM: AQM has been an active area of research ever since RED [18] was proposed. Subsequent work [32, 17, 24] refined the concept and introduced enhancements for stability and fairness. While these schemes reduce queueing delay relative to tail-drop queues, they still induce too large a queueing latency from the ultra-low latency perspective of this paper because they are, fundamentally, queue-based congestion signaling mechanisms. Virtual-queue based algorithms [21, 31] consider signaling congestion based on link utilization. The Phantom Queue is inspired by this work with the difference that PQs are not *adjacent* to physical switch queues. Instead, they operate on network links (in series with switch ports). This makes the PQ completely agnostic to the internal switch architecture and allows it to be deployed with any switch as a 'bump-on-the-wire'.

Transport Layer: Layer 3 research relevant to our ultra-low latency objective includes protocols and algorithms which introduce various changes to TCP or are TCP substitutes. Explicit feedback schemes like XCP [29] and RCP [16] can perform quite well at keeping low queueing but require major features that do not exist in switches or protocols today. Within the TCP family, variants like Vegas [11], CTCP [48] and FAST [53] attempt to control congestion by reacting to increasing RTTs due to queuing delay. By design, such algorithms require the queue to build up to a certain level and, therefore, do not provide ultra-low latency.

Pacing: TCP pacing was suggested for alleviating burstiness due to ACK compression [56]. Support for pacing has not been unanimous. For instance, Aggarwal *et al.* [1] show that paced flows are negatively impacted when competing with non-paced flows because their packets are deliberately delayed. With increasing line rates and the adoption of TCP segmentation offloading, the impact of burstiness has been getting worse [8] and the need for pacing is becoming more evident. One way pacing has been implemented is using software timers to determine when a packet should be transmitted [28, 52, 15]. However, when the ticks of the pacer are very frequent as happens at high line rates, such software-based schemes often lack access to accurate timers or heavily burden the CPU with significant interrupt processing. Further, a software pacer prior to the NIC cannot offset the effects of offloading functions like LSO which occur in the NIC.

8 Final Remarks

In this paper we presented a framework to deliver baseline fabric latency for latency-sensitive applications while simultaneously supporting high fabric goodput for bandwidth-sensitive applications. Through a combination of Phantom Queues to run the network with near zero queueing, adaptive response to ECN marks using DCTCP, and packet pacing to smooth bursts induced by hardware offload mechanisms like LSO, we showed a factor of up to 10-40 reduction in average and tail latency with a configurable sacrifice of overall fabric throughput. Our work makes another case for a time when aggregate bandwidth may no longer be the ultimate evaluation criterion for large-scale fabrics, but a tool in support of other high-level goals such as predictable low latency.

There are two aspects which warrant further investigation. First, it is useful to evaluate HULL on a larger multi-switch testbed and with more diverse workloads. Second, it is important to quantify the buffering requirements for incast-like communication patterns [50] with our approach.

Acknowledgments

We thank our shepherd, Xiaowei Yang, and the anonymous reviewers for their valuable feedback and suggestions. Mohammad Alizadeh is supported by the Caroline and Fabian Pease Stanford Graduate Fellowship.

References

[1] A. Aggarwal, S. Savage, and T. Anderson. Understanding the Performance of TCP Pacing. In *Proc. of INFOCOM*, 2000.

[2] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. In *Proc. of SIGCOMM*, 2008.

[3] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data center TCP (DCTCP). In *Proc. of SIGCOMM*, 2010.

[4] M. Alizadeh, A. Javanmard, and B. Prabhakar. Analysis of DCTCP: stability, convergence, and fairness. In *Proc. of SIGMETRICS*, 2011.

[5] G. Appenzeller, I. Keslassy, and N. McKeown. Sizing router buffers. *Proc. of SIGCOMM*, 2004.

[6] S. Athuraliya, S. Low, V. Li, and Q. Yin. REM: active queue management. *Network, IEEE*, 15(3):48 –53, May 2001.

[7] Cisco Data Center Infrastructure 2.5 Design Guide. http://www.cisco.com/univercd/cc/td/doc/solution/dcidg21.pdf.

[8] N. Beheshti, Y. Ganjali, M. Ghobadi, N. McKeown, and G. Salmon. Experimental study of router buffer sizing. In *Proc. of IMC*, 2008.

[9] T. Benson, A. Anand, A. Akella, and M. Zhang. Understanding data center traffic characteristics. *SIGCOMM Comput. Commun. Rev.*, 40(1):92–99, Jan. 2010.

[10] R. Braden, D. Clark, and S. Shenker. Integrated Services in the Internet Architecture: an Overview, 1994.

[11] L. S. Brakmo, S. W. O'Malley, and L. L. Peterson. TCP Vegas: new techniques for congestion detection and avoidance. *Proc. of SIGCOMM*, 1994.

[12] D. Cohen, T. Talpey, A. Kanevsky, U. Cummings, M. Krause, R. Recio, D. Crupnicoff, L. Dickman, and P. Grun. Remote Direct Memory Access over the Converged Enhanced Ethernet Fabric: Evaluating the Options. In *Proc. of HOTI*, 2009.

[13] DCTCP Linux kernel patch. http://www.stanford.edu/~alizade/Site/DCTCP.html.

[14] A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queueing algorithm. In *Proc. of SIGCOMM*, pages 1–12, 1989.

[15] D.Lacamera. TCP Pacing Linux Implementation. http://danielinux.net/index.php/TCP_Pacing.

[16] N. Dukkipati, M. Kobayashi, R. Zhang-Shen, and N. McKeown. Processor sharing flows in the internet. In *IWQoS*, 2005.

[17] W.-c. Feng, K. G. Shin, D. D. Kandlur, and D. Saha. The BLUE active queue management algorithms. *IEEE/ACM Trans. Netw.*, August 2002.

[18] S. Floyd and V. Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM Trans. Netw.*, 1(4), 1993.

[19] Fulcrum FM4000 Series Ethernet Switch. http://www.fulcrummicro.com/documents/products/FM4000_Product_Brief.pdf.

[20] S. B. Fred, T. Bonald, A. Proutiere, G. Régnié, and J. W. Roberts. Statistical bandwidth sharing: a study of congestion at flow level. In *Proc. of SIGCOMM*, SIGCOMM '01, pages 111–122, 2001.

[21] R. J. Gibbens and F. Kelly. Distributed connection acceptance control for a connectionless network. In *Proc. of the Int'l. Teletraffic Congress*, 1999.

[22] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. VL2: a scalable and flexible data center network. In *Proc. of SIGCOMM*, 2009.

[23] Arista 7100 Series Switches. http://www.aristanetworks.com/en/products/7100series.

[24] C. Hollot, V. Misra, D. Towsley, and W.-B. Gong. On designing improved controllers for AQM routers supporting TCP flows. In *Proc. of INFOCOM*, 2001.

[25] S. Iyer, R. Kompella, and N. McKeown. Designing packet buffers for router linecards. *IEEE/ACM Trans. Netw.*, 16(3):705 –717, june 2008.

[26] V. Jacobson. Congestion avoidance and control. *SIGCOMM Comput. Commun. Rev.*, 18:314–329, August 1988.

[27] H. Jiang and C. Dovrolis. Why is the internet traffic bursty in short time scales. In *In Sigmetrics*, pages 241–252. ACM Press, 2005.

[28] H. Kamezawa, M. Nakamura, J. Tamatsukuri, N. Aoshima, M. Inaba, and K. Hiraki. Inter-Layer Coordination for Parallel TCP Streams on Long Fat Pipe Networks. In *Proc. of SC*, 2004.

[29] D. Katabi, M. Handley, and C. Rohrs. Congestion control for high bandwidth-delay product networks. In *Proc. of SIGCOMM*, 2002.

[30] R. R. Kompella, K. Levchenko, A. C. Snoeren, and G. Varghese. Every microsecond counts: tracking fine-grain latencies with a lossy difference aggregator. In *Proc. of SIGCOMM*, pages 255–266, 2009.

[31] S. S. Kunniyur and R. Srikant. An adaptive virtual queue (AVQ) algorithm for active queue management. *IEEE/ACM Trans. Netw.*, April 2004.

[32] D. Lin and R. Morris. Dynamics of random early detection. In *Proc. of SIGCOMM*, 1997.

[33] The NetFPGA Project. http://netfpga.org.

[34] Cisco Nexus 5548P Switch. http://www.cisco.com/en/US/prod/collateral/switches/ps9441/ps9670/ps11215/white_paper_c11-622479.pdf.

[35] K. Nichols, S. Blake, F. Baker, and D. Black. RFC 2474: Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers.

[36] The Network Simulator NS-2. http://www.isi.edu/nsnam/ns/.

[37] D. Ongaro, S. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum. Fast Crash Recovery in RAMCloud. In *Proc. of SOSP'11*, 2011.

[38] J. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, S. Mitra, A. Narayanan, D. Ongaro, G. Parulkar, M. Rosenblum, S. M. Rumble, E. Stratmann, and R. Stutsman. The case for RAMCloud. *Commun. ACM*, 54:121–130, July 2011.

[39] A. K. Parekh and R. G. Gallager. A generalized processor sharing approach to flow control in integrated services networks: the single-node case. *IEEE/ACM Trans. Netw.*, 1:344–357, June 1993.

[40] K. Ramakrishnan, S. Floyd, and D. Black. RFC 3168: the addition of explicit congestion notification (ECN) to IP.

[41] ConnectX-2 EN with RDMA over Ethernet. http://www.mellanox.com/related-docs/prod_software/ConnectX-2_RDMA_RoCE.pdf.

[42] J. W. Roberts. A survey on statistical bandwidth sharing. *Comput. Netw.*, 45:319–332, June 2004.

[43] B. Schroeder, A. Wierman, and M. Harchol-Balter. Open versus closed: a cautionary tale. In *NSDI'06*, pages 18–18, Berkeley, CA, USA, 2006.

[44] P. Shivam, P. Wyckoff, and D. Panda. EMP: zero-copy OS-bypass NIC-driven gigabit ethernet message passing. In *Proc. of ACM/IEEE conference on Supercomputing*, 2001.

[45] M. Shreedhar and G. Varghese. Efficient fair queueing using deficit round robin. In *Proc. of SIGCOMM*, pages 231–242, 1995.

[46] A. Silberschatz, P. B. Galvin, and G. Gagne. *Operating System Concepts*. Wiley Publishing, 8th edition, 2008.

[47] R. Srikant. *The Mathematics of Internet Congestion Control (Systems and Control: Foundations and Applications)*. 2004.

[48] K. Tan and J. Song. A compound TCP approach for high-speed and long distance networks. In *Proc. IEEE INFOCOM*, 2006.

[49] The tcpdump official website. http://www.tcpdump.org.

[50] V. Vasudevan, A. Phanishayee, H. Shah, E. Krevat, D. G. Andersen, G. R. Ganger, G. A. Gibson, and B. Mueller. Safe and effective fine-grained TCP retransmissions for datacenter communication. In *Proc. of SIGCOMM*, 2009.

[51] A. Vishwanath, V. Sivaraman, and M. Thottan. Perspectives on router buffer sizing: recent results and open problems. *Proc. of SIGCOMM*, 39, 2009.

[52] V. Visweswaraiah and J. Heidemann. Improving restart of idle tcp connections, 1997.

[53] D. X. Wei, C. Jin, S. H. Low, and S. Hegde. FAST TCP: motivation, architecture, algorithms, performance. *IEEE/ACM Trans. Netw.*, 2006.

[54] memcached - a distributed memory object caching system. http://memcached.org.

[55] L. Zhang, S. Deering, D. Estrin, S. Shenker, and D. Zappala. RSVP: a new resource reservation protocol. *Communications Magazine, IEEE*, 40(5):116 –127, May 2002.

[56] L. Zhang, S. Shenker, and D. D. Clark. Observations on the dynamics of a congestion control algorithm: the effects of two-way traffic. *Proc. of SIGCOMM*, 1991.

PACMan: Coordinated Memory Caching for Parallel Jobs

Ganesh Ananthanarayanan [1], Ali Ghodsi [1,4], Andrew Wang [1], Dhruba Borthakur [2], Srikanth Kandula [3], Scott Shenker [1], Ion Stoica [1]

[1] *University of California, Berkeley* [2] *Facebook* [3] *Microsoft Research* [4] *KTH/Sweden*
{ganesha,alig,awang,shenker,istoica}@cs.berkeley.edu, dhruba@facebook.com, srikanth@microsoft.com

Abstract– Data-intensive analytics on large clusters is important for modern Internet services. As machines in these clusters have large memories, in-memory caching of inputs is an effective way to speed up these analytics jobs. The key challenge, however, is that these jobs run multiple tasks in parallel and a job is sped up only when inputs of all such parallel tasks are cached. Indeed, a single task whose input is not cached can slow down the entire job. To meet this "all-or-nothing" property, we have built PACMan, a caching service that coordinates access to the distributed caches. This coordination is essential to improve job completion times and cluster efficiency. To this end, we have implemented two cache replacement policies on top of PACMan's coordinated infrastructure fb–LIFE that minimizes average completion time by evicting large incomplete inputs, and LFU-F that maximizes cluster efficiency by evicting less frequently accessed inputs. Evaluations on production workloads from Facebook and Microsoft Bing show that PACMan reduces average completion time of jobs by 53% and 51% (small interactive jobs improve by 77%), and improves efficiency of the cluster by 47% and 54%, respectively.

1 Introduction

Cluster computing has become a major platform, powering both large Internet services and a growing number of scientific applications. Data-intensive frameworks (*e.g.,* MapReduce [13] and Dryad [20]) allow users to perform data mining and sophisticated analytics, automatically scaling up to thousands of machines.

Machines in these clusters have large memory capacities, which are often underutilized; the median and 95th percentile memory utilizations in the Facebook cluster are 19% and 42%, respectively. In light of this trend, we investigate the use of *memory locality* to speed-up data-intensive jobs by caching their input data.

Data-intensive jobs, typically, have a phase where they process the input data (*e.g., map* in MapReduce [13], *extract* in Dryad [20]). This phase simply reads the raw input and writes out parsed output to be consumed during further computations. Naturally, this phase is IO-intensive. Workloads from Facebook and Microsoft Bing datacenters, consisting of thousands of servers, show that this IO-intensive phase constitutes 79% of a job's duration and consumes 69% of its resources. Our proposal is to speed up these IO-intensive phases by caching their input data in memory. Data is cached after the first access thereby speeding up subsequent accesses.

Using memory caching to improve performance has a long history in computer systems, *e.g.,* [5, 14, 16, 18]. We argue, however, that the *parallel* nature of data-intensive jobs differentiates them from previous systems. Frameworks split jobs in to multiple *tasks* that are run in parallel. There are often enough idle compute slots for small jobs, consisting of few tasks, to run all their tasks in parallel. Such tasks start at roughly the same time and run in a single *wave*. In contrast, large jobs, consisting of many tasks, seldom find enough compute slots to run all their tasks at the same time. Thus, only a subset of their tasks run in parallel.[1] As and when tasks finish and vacate slots, new tasks get scheduled on them. We define the number of parallel tasks as the *wave-width* of the job.

The wave-based execution implies that small single-waved jobs have an *all-or-nothing* property – unless all the tasks get memory locality, there is no improvement in completion time. They run all their tasks in one wave and their completion time is proportional to the duration of the longest task. Large jobs, on the other hand, improve their completion time with every wave-width of their input being cached. Note that the exact set of tasks that run in a wave is not of concern, we only care about the wave-width, *i.e.,* how many of them run simultaneously.

Our position is that *coordinated management* of the distributed caches is required to ensure that enough tasks of a parallel job have memory locality to improve their

[1] We use the terms "small" and "large" jobs to to refer to their input size and/or numbers of tasks.

completion time. Coordination provides a global view that can be used to decide what to evict from the cache, as well as where to place tasks so that they get memory locality. To this end, we have developed PACMan – Parallel All-or-nothing Cache MANager – an in-memory caching system for parallel jobs. On top of PACMan's coordination infrastructure, appropriate placement and eviction policies can be implemented to speed-up parallel jobs.

One such coordinated eviction policy we built, LIFE, aims to *minimize the average completion time of jobs*. In a nutshell, LIFE calculates the wave-width of every job and favors input files of jobs with small waves, *i.e.,* lower wave-widths. It replaces cached blocks of the incomplete file with the largest wave-width. The design of LIFE is driven by two observations. First, a small wave requires caching less data than a large wave to get the same decrease in completion time. This is because the amount of cache required by a job is proportional to its wave-width. Second, we need to retain the entire input of a wave to decrease the completion time. Hence the heuristic of replacing blocks from incomplete files.

Note that maximizing cache hit-ratio – the metric of choice of traditional replacement policies – does not necessarily minimize average completion time, as it ignores the wave-width constraint of parallel jobs. For instance, consider a simple workload consisting of 10 equal-sized single-waved jobs. A policy that caches only the inputs of five jobs will provide a better average completion time, than a policy that caches 90% of the inputs of each job, which will not provide any completion time improvement over the case in which no inputs are cached. However, the first policy will achieve only 50% hit-ratio, compared to 90% hit-ratio for the second policy.

In addition to LIFE, we implemented a second eviction policy, LFU-F, which aims to *maximize the efficiency of the cluster*. Cluster efficiency is defined as finishing the jobs by using the least amount of resources. LFU-F favors popular files and evicts blocks from the least accessed files. Efficiency improves every time data is accessed from cache. So files that are accessed more frequently contribute more to cluster efficiency than files that will be accessed fewer number of times.

A subtle aspect is that the all-or-nothing property is important even for cluster efficiency. This is because tasks of subsequent phases often overlap with the IO-intensive phase. For example, in MapReduce jobs, reduce tasks begin after a certain fraction of map tasks finish. The reduce tasks start reading the output of completed map tasks. Hence a delay in the completion of a few map tasks, when their data is not cached, results in all the reduce tasks waiting. These waiting reduce tasks waste computation slots effectively hurting efficiency.

We have integrated PACMan with Hadoop HDFS [2]. PACMan is evaluated by running workloads from data-

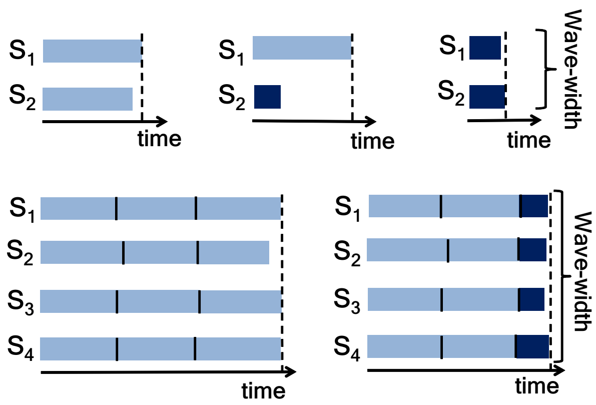


Figure 1: **Example of a single-wave (2 tasks, simultaneously) and multi-wave job (12 tasks, 4 at a time). S_i's are slots. Memory local tasks are dark blocks. Completion time (dotted line) reduces when a wave-width of input is cached.**

centers at Facebook and Bing on an EC2 cluster. We show that overall job completion times reduce by up to 53% with LIFE and cluster efficiency improves by up to 54% with LFU-F. Notably, completion times of small jobs reduce by 77% with LIFE. LIFE and LFU-F outperform traditional schemes like LRU, LFU, and even MIN [11], which is provably optimal for hit-ratios.

2 Cache Replacement for Parallel Jobs

In this section, we first explain how the concept of wave-width is important for parallel jobs, and argue that maximizing cache hit-ratio neither minimizes the average completion time of parallel jobs nor maximizes efficiency of a cluster executing parallel jobs. From first principles, we then derive the ideas behind LIFE and LFU-F cache replacement schemes.

2.1 All-or-Nothing Property

Achieving memory locality for a task will shorten its completion time. But this need not speed up the job. Jobs speed up when an entire wave-width of input is cached (Figure 1). The wave-width of a job is defined as the number of simultaneously executing tasks. Therefore, jobs that consist of a single wave need 100% memory locality to reduce their completion time. We refer to this as the *all-or-nothing* property. Jobs consisting of many waves improve as we incrementally cache inputs in multiples of their wave-width. In Figure 1, the single-waved job runs both its tasks simultaneously and will speed up only if the inputs of both tasks are cached. The multi-waved job, on the other hand, consists of 12 tasks and can run 4 of them at a time. Its completion time improves in steps when any

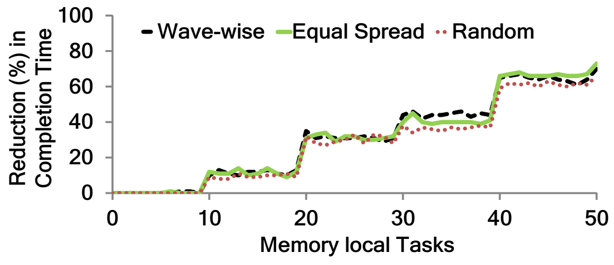


Figure 2: **Reduction in completion time of a job with 50 tasks running on 10 slots. The job speeds up, in steps, when its number of memory local tasks crosses multiples of the wave-width (*i.e.,* 10), regardless of how they are scheduled.**

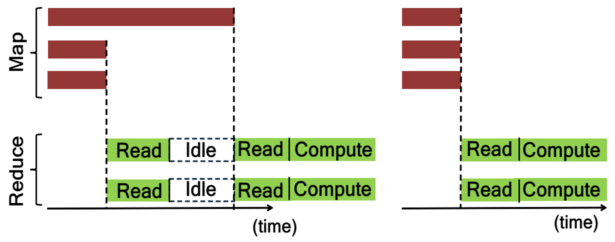


Figure 3: **All-or-nothing property matters for efficiency. In this example of a job with 3 map tasks and 2 reduce tasks, even if one map task is delayed (due to lack of memory locality), reduce tasks idle and hurt efficiency.**

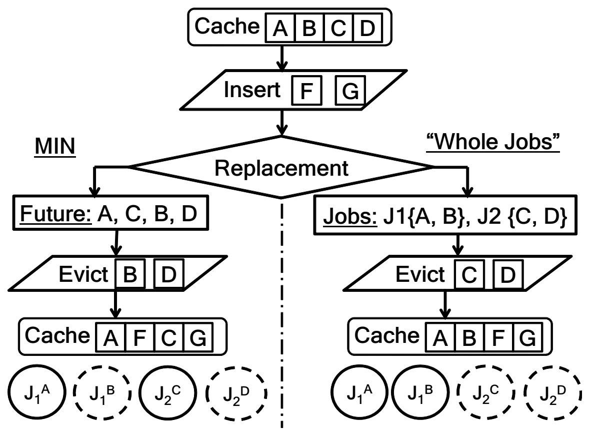


Figure 4: **Cache hit-ratio does not necessarily improve job completion. We consider a cache that has to replace two out of its four blocks. MIN evicts blocks to be accessed farthest in future. "Whole jobs" preserves complete inputs of jobs.**

4, 8 and 12 tasks run memory locally.

We confirmed the hypothesis of wave-widths by executing a sample job on a cluster with 10 slots (see Figure 2). The job operated on 3GB of input and consisted of 50 tasks each working on 60MB of data. Our experiment varied the number of memory-local tasks of the job and measured the reduction in completion time. The baseline was the job running with no caching. Memory local tasks were spread uniformly among the waves ("Equal Spread"). We observed the job speeding up when its number of memory-local tasks crossed 10, 20 and so forth, *i.e.,* multiples of its wave-width, thereby verifying the hypothesis. Further, we tried two other scheduling strategies. "Wave-wise" scheduled the non-memory-local tasks before memory local tasks, *i.e.,* memory local tasks ran simultaneously, and "Random" scheduled the memory local tasks in an arbitrary order. We see that the speed-up in steps of wave-width holds in both cases, albeit with slightly reduced gains for "Random". Surprisingly, the wave-width property holds even when memory local tasks are randomly scheduled because a task is allotted a slot only when slots become vacant, not a priori. This automatically balances memory local tasks and non-memory-local tasks across the compute slots.

That achieving memory locality will lower resource usage is obvious – tasks whose inputs are available in memory run faster and occupy the cluster for fewer hours. A subtler point is that the all-or-nothing constraint can also be important for cluster efficiency. This is because some of the schedulers in parallel frameworks (*e.g.,* Hadoop and MPI) allow tasks of subsequent stages to begin even before all tasks in the earlier stages finish. Such "pipelining" can hide away some data transfer latency, for example, when reduce tasks start running even before the last task in the map stage completes [3]. However, this means that a delay in the completion of some map tasks, perhaps due to lack of memory locality, results in all the reduce tasks waiting. These waiting reduce tasks waste computation slots and adversely affect efficiency. Figure 3 illustrates this overlap with an example job of three map tasks and two reduce tasks.

In summary, meeting the all-or-nothing constraint improves completion time and efficiency of parallel jobs.

2.2 Sticky Policy

Traditional cache replacement schemes that maximize cache hit-ratio do not consider the wave-width constraint of all-or-nothing parallel jobs. Consider the situation depicted in Figure 4 of a 4-entry cache storing blocks *A*, *B*, *C* and *D*. Job J_1's two tasks will access blocks *A* and *B*, while job J_2's two tasks will access *C* and *D*. Both jobs consist of just a single wave and hence their job completion time improves only if their entire input is cached.

Now, pretend a third job J_3 with inputs *F* and *G* is scheduled before J_1 and J_2, requiring the eviction of two blocks currently in the cache. Given the oracular knowledge that the future block access pattern will be *A*, *C*, *B*, then *D*, MIN [11] will evict the blocks accessed farthest in the future: *B* and *D*. Then, when J_1 and J_2 execute, they both experience a cache miss on one of their tasks. These cache misses bound their completion times, meaning that MIN cache replacement resulted in no reduction in completion time for either J_1 or J_2. Consider an alternate replacement scheme that chooses to evict the input set of J_2 (C and D). This results in a reduction in completion time for J_1 (since its entire input set of A and B is cached). J_2's completion time is unaffected. Note that the cache hit-ratio remains the same as for MIN (50%).

Further, maximizing hit-ratio does not maximize efficiency of the cluster. In the same example as in Figure 4, let us add a reduce task to each job. Both J_1 and J_2 have two map tasks and one reduce task. Let the reduce task start after 5% of the map tasks have completed (as in Hadoop [3]). We now compare the resource consumption of the two jobs with MIN and "whole jobs" which evicts inputs of J_2. With MIN, the total resource consumption is $2(1+\mu)m + 2(0.95)m$, where m is the duration of a non-memory-local task and μ reflects the speed-up when its input is cached; we have omitted the computation of the reduce task. The policy of "whole jobs", on the other hand, expends $2(1+\mu)m + (0.95\mu + 0.05)m$ resources. As long as memory locality produces a speed-up, *i.e.,* $\mu \leq 1$, MIN consumes more resources.

The above example, in addition to illustrating that cache hit-ratios are insufficient for both speeding up jobs and improving cluster efficiency, also highlights the importance of retaining *complete* sets of inputs. Improving completion time requires retaining complete wave-widths of inputs, while improving efficiency requires retaining complete inputs of jobs. Note that retaining the complete inputs of jobs automatically meets the wave-width constraint to reduce completion times. Therefore, instead of evicting the blocks accessed farthest in the future, replacement schemes for parallel jobs should recognize the commonality between inputs of the same job and evict at the granularity of a job's input.

This intuition gives rise to the *sticky* policy. *The sticky policy preferentially evicts blocks of incomplete files.* If there is an incomplete file in cache, it sticks to its blocks for eviction until the file is completely removed from cache. The sticky policy is crucial as it disturbs the fewest completely cached inputs and evicts the incomplete files which are not beneficial for jobs.[2]

Given the sticky policy to achieve the all-or-nothing requirement, we now address the question of which inputs to retain in cache such that we minimize average completion time of jobs and maximize cluster efficiency.

2.3 Average Completion Time – LIFE

We show that in a cluster with multiple jobs, favoring jobs with the smallest wave-widths minimizes the average completion time of jobs. Assume that all jobs in the cluster are single-waved. Every job j has a wave-width of w and an input size of I. Let us assume the input of a job is equally distributed among its tasks. Each task's input size is $\left(\frac{I}{w}\right)$ and its duration is proportional to its input size. As before, memory locality reduces its duration by a factor of μ. The factor μ is dictated by the difference between memory and disk bandwidths, but limited by additional overheads such as deserialization and decompression of the data after reading it.

To speed up a single-waved job, we need I units of cache space. On spending I units of cache space, tasks would complete in $\mu\left(\frac{I}{w}\right)$ time. Therefore the saving in completion time would be $(1-\mu)\left(\frac{I}{w}\right)$. Counting this savings for every access of the file, it becomes $f(1-\mu)\left(\frac{I}{w}\right)$, where f is the frequency of access of the file. Therefore, the ratio of the job's benefit to its cost is $f(1-\mu)\left(\frac{1}{w}\right)$. In other words, it is directly proportional to the frequency and inversely proportional to the wave-width. The smaller the wave-width, the larger the savings in completion time per unit of cache spent. This is illustrated in Figure 5 comparing two jobs with the same input size (and of the same frequency), but wave-widths of 2 and 4. Clearly, it is better to use I units of cache space to store the input of the job with a wave-width of two. This is because its work per task is higher and so the savings are proportionately more. Note that even if the two inputs are unequal (say, I_1 and I_2, and $I_1 > I_2$), caching the input of the job with lower wave-width (I_1) is preferred despite its larger input size. Therefore, in a cluster with multiple jobs, *average completion time is best reduced by favoring the jobs with smallest wave-widths* (LIFE).

This can be easily extended to a multi-waved jobs. Let the job have n waves, c of which have their inputs cached. This uses $cw\left(\frac{I}{nw}\right)$ of cache space. The benefit in completion time is $f(1-\mu)c\left(\frac{I}{nw}\right)$. The ratio of the job's benefit to its cost is $f(1-\mu)\left(\frac{1}{w}\right)$, hence best reduced by still picking the jobs that have the smallest wave-widths.

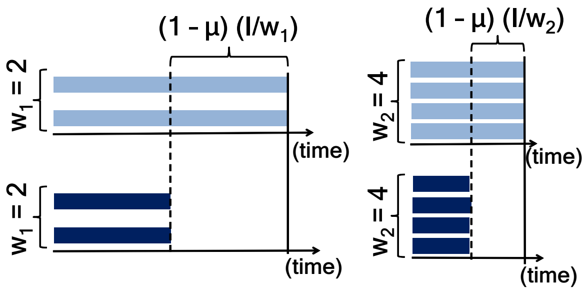


Figure 5: **Gains in completion time due to caching decreases as wave-width increases. Solid and dotted lines show completion times without and with caching (for two jobs with input of I but wave-widths of 2 and 4). Memory local tasks are dark blocks, sped up by a factor of μ.**

[2]When there are strict barriers between phases, the sticky policy does not improve efficiency. Nonetheless, such barriers are akin to "application-level stickiness" and hence stickiness at the caching layer beneath, expectedly, does not add value.

	Facebook	Microsoft Bing
Dates	Oct 2010	May-Dec* 2009
Framework	Hadoop	Dryad
File System	HDFS [2]	Cosmos
Script	Hive [4]	Scope [24]
Jobs	375K	200K
Input Data	150PB	310PB
Cluster Size	3,500	Thousands
Memory per machine	48GB	N/A

* *One week in each month*

Table 1: **Details of Hadoop and Dryad datasets analyzed from Facebook and Microsoft Bing clusters, respectively.**

2.4 Cluster Efficiency – LFU-F

In this section, we derive that retaining frequently accessed files maximizes efficiency of the cluster. We use the same model for the cluster and its jobs as before. The cluster consists of single-waved jobs, and each job j has wave-width w and input size I. Duration of tasks are proportional to their input sizes, $\left(\frac{I}{w}\right)$, and achieving memory locality reduces its duration by a factor of μ.

When the input of this job is cached, we use I units of cache. In return, the savings in efficiency is $(1-\mu)I$. The savings is obtained by summing the reduction in completion time across all the tasks in the wave, *i.e.,* $w \cdot (1-\mu)\left(\frac{I}{w}\right)$. Every memory local task contributes to improvement in efficiency. Further, the savings of $(1-\mu)I$ is obtained on every access of the file, thereby making its aggregate value $f(1-\mu)I$ where f is the frequency of access of the file. Hence, the ratio of the benefit to every unit of cache space spent on this job is $f(1-\mu)$, or a function of only the frequency of access of the file. Therefore, *cluster efficiency is best improved by retaining the most frequently accessed files* (LFU-F).

This naturally extends to multi-waved jobs. As jobs in data-intensive clusters typically read entire files, frequency of access of inputs across the different waves of a job is the same. Hence cluster efficiency is best improved by still favoring the frequently accessed files.

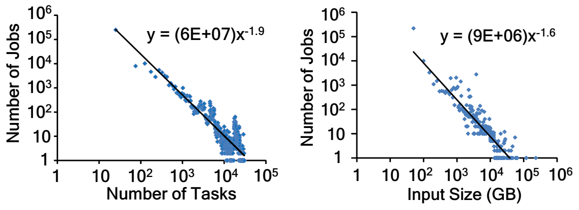
To summarize, we have shown that, (i) the all-or-nothing property is crucial for improving completion time of jobs as well as efficiency, (ii) average completion time is minimized by retaining inputs of jobs with low wave-widths, and (iii) cluster efficiency is maximized by retaining the frequently used inputs. We next show some relevant characteristics from production workloads, before moving on to explain the details of PACMan.

3 Workloads in Production Clusters

In this section, we analyze traces from two production clusters, each consisting of thousands of machines – Facebook's Hadoop cluster and Microsoft Bing's Dryad cluster. Together, they account over half a million jobs processing more than 400PB of data. Table 1 lists the relevant details of the traces and the clusters.

All three clusters co-locate storage and computation. The distributed file systems in these clusters store data in units of *blocks*. Every task of a job operates on one or more *blocks* of data. For each of the jobs we obtain task-level information: start and end times, size of the blocks the task reads (local or remote) and writes, the machine the task runs on, and the locations of its inputs.

Our objective behind analyzing these traces is to highlight characteristics – heavy tail distribution of input sizes of jobs, and correlation between file size and popularity – that are relevant for LIFE and LFU-F.



(a) Number of tasks (b) Input Size

Figure 6: **Power-law distribution of jobs (Facebook) in the number of tasks and input sizes. Power-law exponents are 1.9 and 1.6 when fitted with least squares regression.**

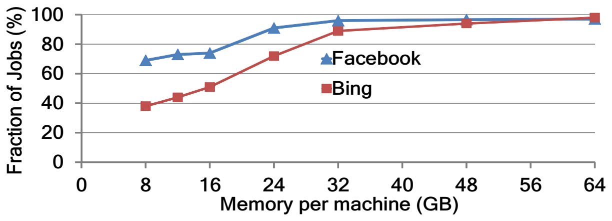


Figure 7: **Fraction of active jobs whose data fits in the aggregate cluster memory, as the memory per machine varies.**

3.1 Heavy-tailed Input Sizes of Jobs

Datacenter jobs exhibit a heavy-tailed distribution of input sizes. Workloads consist of many small jobs and relatively few large jobs. In fact, 10% of overall data read is accounted by a disproportionate 96% and 90% of the smallest jobs in the Facebook and Bing workloads. As Figure 6 shows, job sizes – input sizes and number of tasks – indeed follow a power-law distribution, as the log-log plot shows a linear relationship.

The skew in job input sizes is so pronounced that a large fraction of active jobs can simultaneously fit their entire data in memory.[3] We perform a simple simulation that looks at jobs in the order of their arrival time. The simu-

[3] By active jobs we mean jobs that have at least one task running.

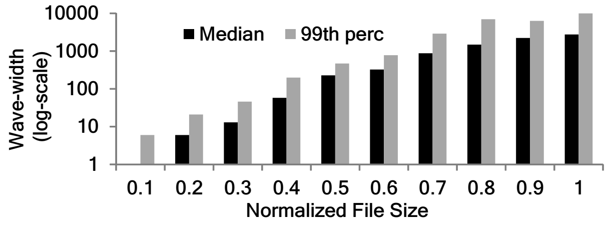


Figure 8: **Wave-width, *i.e.,* number of simultaneous tasks, of jobs as a function of sizes of files accessed. File sizes are normalized to the largest file; the largest file has size 1.**

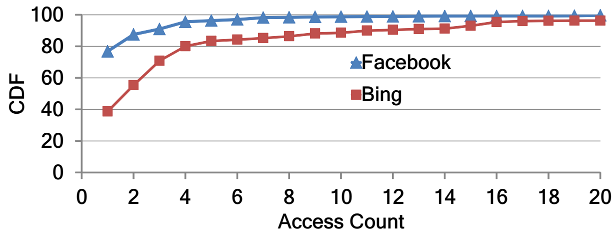


Figure 9: **Skewed popularity of data. CDF of the access counts of the input blocks stored in the cluster.**

lator assumes the memory and computation slots across all the machines in the cluster to be aggregated. It loads a job's entire input into memory when it starts and deletes it when the job completes. If the available memory is insufficient for a job's entire input, none of it is loaded. Figure 7 plots the results of the simulation. For the workloads from Facebook and Bing, we see that 96% and 89% of the active jobs respectively can have their data entirely fit in memory, given an allowance of 32GB memory per server for caching. This bodes well for satisfying the all-or-nothing constraint of jobs, crucial for the efficacy of LIFE and LFU-F.

In addition to being easier to fit a small job's input in memory, its wave-width is smaller. In our workloads, wave-widths roughly correlate with the input file size of the job. Figure 8 plots the wave-width of jobs binned by the size of their input files. Small jobs, accessing the smaller files, have lower wave-widths. This is because, typically, small jobs do not have sufficient number of tasks to utilize the slots allocated by the scheduler. This correlation helps to explore an approximation for LIFE to use file sizes instead of estimating wave-widths (§4.2).

3.2 Large Files are Popular

Now, we look at popularity skew in data access patterns. As noted in prior work, the popularity of input data is skewed in data-intensive clusters [15]. A small fraction of the data is highly popular, while the rest is accessed less frequently. Figure 9 shows that the top 12% of popular data is accessed 10× more than the bottom third in the Bing cluster. The Facebook cluster demonstrates a similar skew. The top 5% of the blocks are seven times more popular than the bottom three-quarters.

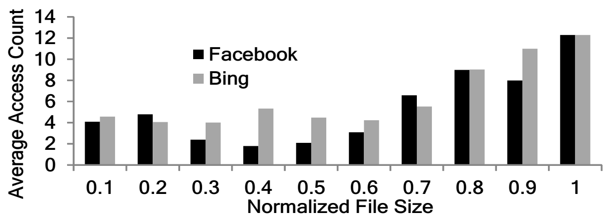


Figure 10: **Access count of files as a function of their sizes, normalized to the largest file; largest file has size 1. Large files, accessed by production jobs, have higher access count.**

Interestingly, large files have higher access counts (see Figure 10). Often they are accessed by production jobs to generate periodic (hourly) summaries, *e.g.,* financial and performance metrics, from various large logs over consolidated time intervals in the past. These intervals could be as large as weeks and months, directly leading to many of the logs in that interval being repeatedly accessed. The popularity of large files, whose jobs consume most resources, strengthens the idea from §2.4 that favoring frequently accessed files is best for cluster efficiency.

We also observe repeatability in the data accesses. *Single-accessed files are spread across only 11% and 6% of jobs* in the Facebook and Bing workloads. Even in these jobs, not all the data they access is singly-accessed. Hence, we have sufficient repeatability to improve job performance by caching their inputs.

4 PACMan: System Design

We first present PACMan's architecture that enables the implementation of the sticky policy, and then discuss the details involved in realizing LIFE and LFU-F.

4.1 Coordination Architecture

PACMan globally coordinates access to its caches. Global coordination ensures that a job's different input blocks, distributed across machines, are viewed in unison to satisfy the all-or-nothing constraint. To that end, the two requirements from PACMan are, (*a*) support queries for the set of machines where a block is cached, and (*b*) mediate cache replacement globally across the machines.

PACMan's architecture consists of a central *coordinator* and a set of *clients* located at the storage nodes of the cluster (see Figure 11). Blocks are added to the PACMan clients. PACMan clients update the coordinator when the state of their cache changes (*i.e.,* when a block is added or removed). The coordinator uses these updates to maintain a mapping between every cached block and the ma-

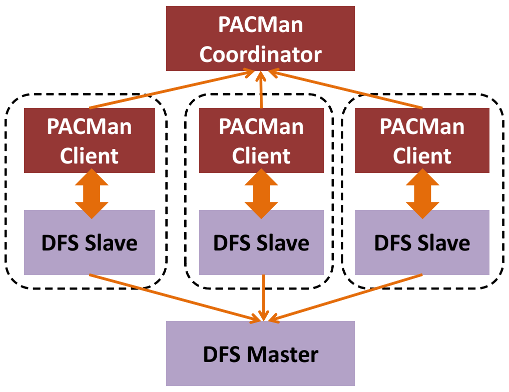


Figure 11: PACMan **architecture. The central *coordinator* manages the distributed *clients*. Thick arrows represent data flow while thin arrows denote meta-data flow.**

chines that cache it. As part of the map, it also stores the file that a block belongs to and the wave-width of jobs when accessing that file (§4.2). This global map is leveraged by LIFE and LFU-F in implementing the sticky policy to look for incomplete files. Frameworks work with the coordinator to achieve memory locality for tasks.

The client's main role is to serve cached blocks, as well as cache new blocks. We choose to cache blocks at the *destination*, *i.e.,* the machine where the task executes as opposed to the *source*, *i.e.,* the machine where the input is stored. This allows an uncapped number of replicas in cache, which in turn increases the chances of achieving memory locality especially when there are hotspots due to popularity skew [15]. Memory local tasks contact the local PACMan client to check if its input data is present. If not, they fetch it from the distributed file system (DFS). If the task reads data from the DFS, it puts it in cache of the local PACMan client and the client updates the coordinator about the newly cached block. Data flow is designed to be local in PACMan as remote memory access could be constrained by the network.

Fault Tolerance: The coordinator's failure does not hamper the job's execution as data can always be read from disk. However, we include a secondary coordinator that functions as a cold standby. Since the secondary coordinator has no cache view when it starts, clients periodically send updates to the coordinator informing it of the state of their cache. The secondary coordinator uses these updates to construct the global cache view. Clients do not update their cache when the coordinator is down.

Scalability: Nothing precludes distributing the central coordinator across different machines to avoid having it be a bottleneck. We have, however, found that the scalability of our system suffices for our workloads (see §5.6).

4.2 Wave-width

Wave-width is important for LIFE as it aims to retain inputs of jobs with lower wave-widths. However, both defining and calculating wave-widths is non-trivial because tasks do not strictly follow wave boundaries. Tasks get scheduled as and when previous tasks finish and slots become available. This makes modeling the tasks that run in a wave complicated. Slots also open up for scheduling when the scheduler allots extra slots to a job during periods of low utilization in the cluster. Therefore, wave-widths are not static during a job's execution. They are decided based on slot availabilities, fairness restrictions and scheduler policies. Unlike MIN, which is concerned only with the *order* in which requests arrive, our setting requires knowing the exact time of the request, which in turn requires estimating the speed-up due to memory locality for each task. All these factors are highly variable and hard to model accurately.

Given such a fluid model, we propose the following approximation. We make periodic measurements of the number of concurrent tasks of a job. When a job completes, we get a set of values, $\langle(w, t(w))\rangle$, such that $0 < t(w) \leq 1$. For every value of wave-width, w, $t(w)$ shows the fraction of time spent by the job with that wave-width. We take measurements every 1s in practice.

Note that while $\sum t(w) = 1$, $\sum w$ is not necessarily equal to the number of tasks in the job. This is because wave boundaries are not strict and tasks overlap between measurements. This led us to drop the idea of using the measurements of $\langle(w, t(w))\rangle$ to divide blocks of a file into different *waves*. Also, such an explicit division requires the scheduler to collectively schedule the tasks operating on a wave. Therefore, despite the potential benefits, to sidestep the above problems, we assign a single value for the wave-width of a file. We define the wave-width of a file as a weighted average across the different observed wave-widths, $\sum w \cdot t(w)$. The wave-width is included in the mapping maintained by the coordinator.

A noteworthy approximation to wave-widths is to simply consider small and large jobs instead, based on their input sizes. As Figure 8 showed, there is a correlation between input sizes of jobs and their wave-widths. Therefore, such an approximation mostly maintains the relative ordering between small and large waves despite approximating them to small and large job input sizes. We evaluate this approximation in §5.3.

4.3 LIFE and LFU-F within PACMan

We now describe how LIFE and LFU-F are implemented inside PACMan's coordinated architecture.

The coordinated infrastructure's global view is fundamental to implementing the sticky policy. Since LIFE and

```
procedure FileToEvict_LIFE(Client c)
    cFiles = fileSet.filter(c)                         ▷ Consider only c's files
    f = cFiles.olderThan(window).oldest()              ▷ Aging
    if f == null then                                  ▷ No old files to age out
        f = cFiles.getLargestIncompleteFile()
    if f == null then                                  ▷ Only complete files left
        f = cFiles.getLargestCompleteFile()
    return f.name                                      ▷ File to evict

procedure FileToEvict_LFU-F(Client c)
    cFiles = fileSet.filter(c)                         ▷ Consider only c's files
    f = cFiles.olderThan(window).oldest()              ▷ Aging
    if f == null then                                  ▷ No old files to age out
        f = cFiles.getLeastAccessedIncompleteFile()
    if f == null then                                  ▷ Only complete files left
        f = cFiles.getLeastAccessedCompleteFile()
    return f.name                                      ▷ File to evict

procedure Add(Client c, String name, Block bId)
    File f = fileSet.getByName(name)
    if f == null then
        f = new File(name)
        fileSet.add(f)
    f.addLocation(c, bId)                              ▷ Update properties
```

Pseudocode 1: **Implementation of LIFE and LFU-F – from the perspective of the PACMan coordinator.**

LFU-F are global cache replacement policies, they are implemented at the coordinator. Pseudocode 1 describes the steps in implementing LIFE and LFU-F. In the following description, we use the terms file and input interchangeably. If all blocks of a file are cached, we call it a *complete file*; otherwise it is an *incomplete file*. When a client runs out of cache memory it asks the coordinator for a file whose blocks it can evict, by calling FileToEvict() (LIFE or LFU-F).

To make this decision, LIFE first checks whether the client's machine caches the blocks of any incomplete file. If there are many such incomplete files, LIFE picks the one with the largest wave-width and returns it to the client. There are two points worth noting. First, by picking an incomplete file, LIFE ensures that the number of fully cached files does not decrease. Second, by picking the largest incomplete file, LIFE increases the opportunity for more small files to remain in cache. If the client does not store the blocks of any incomplete file, LIFE looks at the list of complete files whose blocks are cached by the client. Among these files, it picks the one with the largest wave-width. This increases the probability of multiple small files being cached in future.

LFU-F rids the cache of less frequently used files. It assumes that the future accesses of a file is predicted by its current frequency of access. To evict a block, it first checks if there are incomplete files and picks the *least accessed* among them. If there are no incomplete files, it picks the complete file with the smallest access count.

To avoid cache pollution with files that are never evicted, we also implement a window based aging mechanism. Before checking for incomplete and complete files, both LIFE and LFU-F check whether the client stores any blocks of a file that has not been referred to for at least *window* time period. Among these files, it picks the one that has been accessed the least number of times. This makes it flush out the aged and less popular blocks. In practice, we set the window to be large (*e.g.,* hours), and it has had limited impact on most workloads.

PACMan operates in conjunction with the DFS. However, in practice, they are insulated from the job identifiers that access them. Therefore, we approximate the policy of maximizing the number of whole job inputs to maximizing the number of whole *files* that are present in cache, an approximation that works well (§5.3).

Finally, upon caching a block, a client contacts the coordinator by calling Add(). This allows the coordinator to maintain a global view of the cache, including the access count for every file for implementing LFU-F. Similarly, when a block is evicted, the client calls Remove() to update the coordinator's global view. We have omitted the pseudocode for this for brevity.

Pluggable policies: PACMan's architecture is agnostic to replacement algorithms. Its global cache view can support any replacement policy that needs coordination.

5 Evaluation

We built PACMan and modified HDFS [2] to leverage PACMan's caching service. The prototype is evaluated on a 100-node cluster on Amazon EC2 [1] using workloads derived from the Facebook and Bing traces (§3). To compare at a larger scale against a wider set of idealized caching techniques, we use a trace-driven simulator that performs a detailed replay of task logs. We first describe our evaluation setup before presenting our results.

5.1 Setup

Workload: Our workloads are derived from the Facebook and Bing traces described in §3, representative of Hadoop and Dryad systems. The key goals during this derivation was to preserve the original workload's characteristics, specifically the heavy-tailed nature of job input sizes (§3.1), skewed popularity of files (§3.2), and load proportional to the original clusters.

We meet these goals as follows. We replay jobs with the same inter-arrival times and input files as in the original workload. However, we scale down the file sizes proportionately to reflect the smaller size of our cluster and, consequently, reduced aggregate memory. Thereby, we en-

Bin	Tasks	% of Jobs		% of Resources	
		Facebook	Bing	Facebook	Bing
1	1–10	85%	43%	8%	6%
2	11–50	4%	8%	1%	5%
3	51–150	8%	24%	3%	16%
4	151–500	2%	23%	12%	18%
5	> 500	1%	2%	76%	55%

Table 2: **Job size distributions. The jobs are binned by their sizes in the scaled-down Facebook and Bing workloads.**

sure that there is sufficient memory for the same fraction of jobs' input as in the original workload. Overall, this helps us mimic the load experienced by the original clusters as well as the access patterns of files. We confirmed by simulation (described shortly) that performance improvements with the scaled down version matched that of the full-sized cluster.

Cluster: We deploy our prototype on 100 Amazon EC2 nodes, each of them "double-extra-large" machines [1] with 34.2GB of memory, 13 cores and 850GB of storage. PACMan is allotted 20GB of cache per machine; we evaluate PACMan's sensitivity to cache space in §5.5.

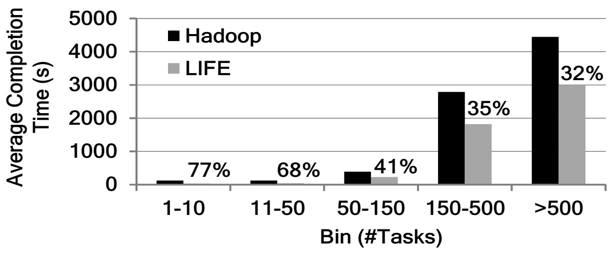
Trace-driven Simulator: We use a trace-driven simulator to evaluate PACMan at larger scales and longer durations. The simulator performed a detailed and faithful replay of the task-level traces of Hadoop jobs from Facebook and Dryad jobs from Bing. It preserved the read/write sizes of tasks, replica locations of input data as well as job characteristics of failures, stragglers and recomputations [7]. The simulator also mimicked fairness restrictions on the number of permissible concurrent slots, a key factor for the number of waves in the job.

We use the simulator to test PACMan's performance at the scale of the original datacenters, as well as to mimic ideal cache replacement schemes like MIN.

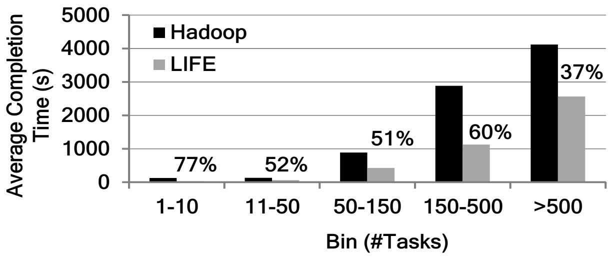
Metrics: We evaluate PACMan on two metrics that it optimizes – average completion time of jobs and efficiency of the cluster. The baseline for our deployment is Hadoop-0.20.2. The trace-driven simulator compared with currently deployed versions of Hadoop and Dryad.

Job Bins: To separate the effect of PACMan's memory locality on different jobs, we binned them by the number of map tasks they contained in the scaled-down workload. Table 2 shows the distribution of jobs by count and resources. The Facebook workload is dominated by small jobs – 85% of them have ≤ 10 tasks. The Bing workload, on the other hand, has the corresponding fraction to be smaller but still sizable at 43%. When viewed by the resources consumed, we obtain a different picture. Large jobs (bin-5), that are only 1% and 2% of all jobs, consume a disproportionate 76% and 55% of all resources. The skew between small and large jobs is higher in the Facebook workload than in the Bing workload.

The following is a summary of our results.



(a) Facebook Workload



(b) Bing Workload

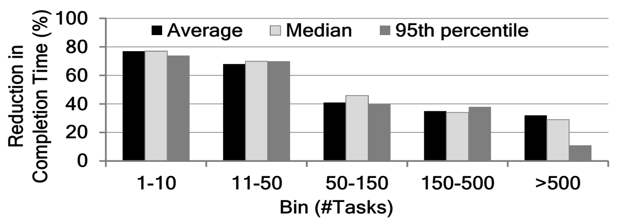
Figure 12: **Average completion times with LIFE, for Facebook and Bing workloads. Relative improvements compared to Hadoop are marked for each bin.**

- Average completion times improve by 53% with LIFE; small jobs improve by 77%. Cluster efficiency improves by 54% with LFU-F (§5.2).
- Without the *sticky* policy of evicting from incomplete files, average completion time is 2× more and cluster efficiency is 1.3× worse (§5.3).
- LIFE and LFU-F are better than MIN in improving job completion time and cluster efficiency, despite a lower cache hit-ratio (§5.4).

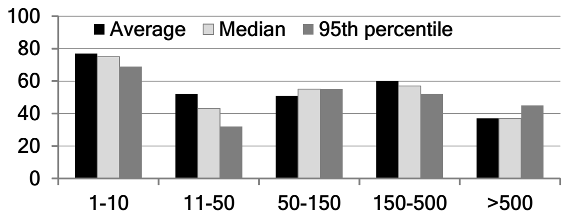
We evaluate our replacement schemes before describing the impact of systemic factors (§5.5 and §5.6).

5.2 PACMan's Improvements

LIFE improves the average completion time of jobs by 53% and 51% in the two workloads. As Figure 12 shows small jobs (bin-1 and bin-2) benefit considerably. Jobs in bin-1 see their average completion time reduce by 77% with the gains continuing to hold in bin-2. As a direct consequence of the sticky policy, 74% of jobs in the Facebook workload and 48% of jobs in the Bing workload meet the all-or-nothing constraint, *i.e.,all* their tasks being memory local. Large jobs benefit too (bin-5) seeing an improvement of 32% and 37% in the two workloads. This highlights LIFE's automatic adaptability. While it favors small jobs, the benefits automatically spill over to large jobs when there is spare cache space.



(a) Facebook Workload



(b) Bing Workload

Figure 13: **Distribution of gains for Facebook and Bing workloads. We present the improvement in average, median and 95th percentile completion times.**

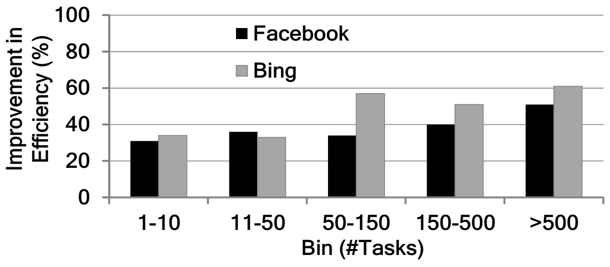


Figure 14: **Improvement in cluster efficiency with LFU-F compared to Hadoop. Large jobs contribute more to improving efficiency due to their higher frequency of access.**

Testbed	Scale	LIFE		LFU-F	
		Facebook	Bing	Facebook	Bing
EC2	100	53%	51%	47%	54%
Simulator	1000's*	55%	46%	43%	50%

** Original cluster size*

Table 3: **Summary of results. We list improvement in completion time with LIFE and cluster efficiency with LFU-F.**

Figure 13 elaborates on the distribution – median and 95th percentile completion times – of LIFE's gains. The encouraging aspect is the tight distribution in bin-1 with LIFE where the median and 95th percentile values differ by at most 6% and 5%, respectively. Interestingly, the Facebook results in bin-2 and the Bing results in bin-3 are spread tighter compared to the other workload.

LFU-F improves cluster efficiency by 47% with the Facebook workload and 54% with the Bing workload. In large clusters, this translates to significant room for executing more computation. Figure 14 shows how this gain in efficiency is derived from different job bins. Large jobs have a higher contribution to improvement in efficiency than the small jobs. This is explained by the observation in §3.2 that large files are more frequently accessed.

An interesting question is the effect LIFE and LFU-F have on the metric that is not their target. With LIFE in deployment, cluster efficiency improves by 41% and 43% in the Facebook and Bing workloads. These are comparable to LFU-F because of the power-law distribution of job sizes. Since small jobs require only a little cache space, even after favoring their inputs, there is space remaining for the large (frequently accessed) files. However, the power-law distribution results in LFU-F's poor performance on average completion time. Average completion time of jobs improves by only 15% and 31% with LFU-F. Favoring the large frequently accessed files leaves insufficient space for small jobs, whose improvement has the highest impact on average completion time.

Overall, we see that memory local tasks run 10.8× faster than those that read data from disk.

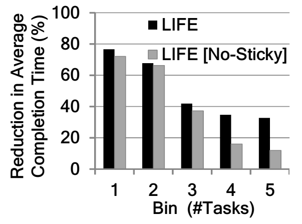
Simulation: We use the trace-driven simulator to assess PACMan's performance on a larger scale of thousands of machines (same size as in the original clusters). The simulator uses 20GB of memory per machine for PACMan's cache. LIFE improves the average completion times of jobs by 58% and 48% for the Facebook and Bing workloads. LFU-F's results too are comparable to our EC2 deployment with cluster efficiency improving by 45% and 51% in the two workloads. This increases our confidence in the large-scale performance of PACMan as well as our methodology for scaling down the workload. Table 3 shows a comparative summary of the results.

5.3 LIFE and LFU-F

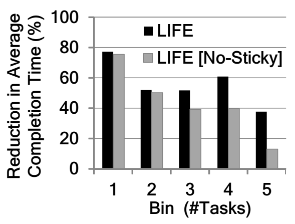
In this section, we study different aspects of LIFE and LFU-F. The aspects under focus are the sticky policy, approximation of wave-widths to file size, and using whole file inputs instead of whole job inputs.

Sticky Policy: An important aspect of LIFE and LFU-F is its sticky policy that prefers to evict blocks from already incomplete files. We test its value with two schemes, LIFE[No-Sticky] and LFU-F[No-Sticky]. LIFE[No-Sticky] and LFU-F[No-Sticky] are simple modifications to LIFE and LFU-F to not factor the incompleteness while evicting. LIFE[No-Sticky] evicts the file with the largest wave-width in the cache, LFU-F[No-Sticky] just evicts blocks from the least frequently accessed file. Ties are broken arbitrarily.

Figure 15 compares LIFE[No-Sticky] with LIFE. Large jobs are hurt most. The performance of LIFE[No-Sticky] is 2× worse than LIFE in bin-4 and 3× worse in bin-5. Interestingly, jobs in bin-1 are less affected. While LIFE and

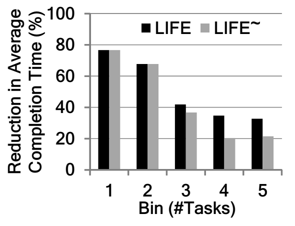


(a) Facebook

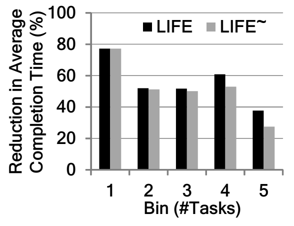


(b) Bing

Figure 15: **Sticky policy. LIFE [No-Sticky] evicts the largest file in cache, and hence is worse off than LIFE.**

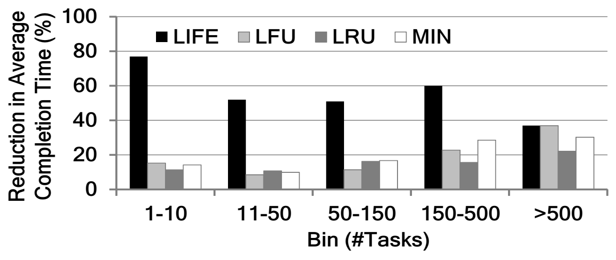


(a) LIFE~ vs. LIFE – Facebook

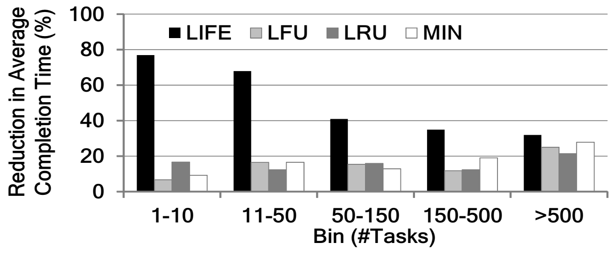


(b) LIFE~ vs. LIFE – Bing

Figure 16: **Approximating LIFE to use file sizes instead of wave-widths. Accurately estimating wave-widths proves important for large jobs.**



(a) Facebook Workload



(b) Bing Workload

Figure 17: **Comparison between LIFE, LFU-F, LFU, LRU and MIN cache replacements.**

LIFE[No-Sticky] differ in the way they evict blocks of the large files, there are enough large files to avoid disturbing the inputs of small jobs.

LFU-F[No-Sticky]'s improvement in efficiency is 32% and 39% for the two workloads, sharply reduced values in contrast to LFU-F's 47% and 54% with the Facebook and Bing workloads. These results strongly underline the value of coordinated replacement in PACMan by looking at global view of the cache.

Wave-width vs. File Sizes: As alluded to in §4.2, we explore the benefits of using file sizes as a substitute for wave-width. The intuition is founded on the observation in §3.1 (Figure 8) that wave-widths roughly correlate with file sizes. We call the variant of LIFE that uses file sizes as LIFE~. The results in Figure 16 shows that while LIFE~ keeps up with LIFE for small jobs, there is significant difference for the large jobs in bin-4 and bin-5. The detailed calculation of the wave-widths pays off with improvements differing by 1.7× for large jobs.

Whole-jobs: Recall from §2.2 and §4.3 that our desired policy is to retain inputs of as many whole job inputs as possible. As job-level information is typically not available at the file system or caching level, for ease and cleanliness of implementation, LIFE and LFU-F approximate this by retaining as many whole *files* as possible.

Evaluation shows that LIFE is on par with the eviction policy that retains whole job inputs, for small jobs. This is because small jobs typically tend to operate on single files, therefore the approximation does not introduce errors. For larger jobs, that access multiple files, LIFE takes a 11% hit in performance. The difference due to LFU-F using whole files instead of whole job inputs is just a 2% drop in efficiency. The comparable results make us conclude that the approximation is a reasonable trade-off for the significant implementation ease.

5.4 Traditional Cache Replacement

Our prototype also implements traditional cache replacement techniques like LRU and LFU. Figure 17 and Table 4 compare LIFE's performance. Table 5 contrasts LFU-F's performance. LIFE outperforms both LRU and LFU for small jobs while achieving comparable performance for large jobs. Likewise, LFU-F is better than LRU and LFU in improving cluster efficiency.

Interestingly, LIFE and LFU-F outperform even MIN [11], the optimal replacement algorithm for cache hit-ratio. MIN deletes the block that is to be accessed farthest in the future. As Figure 17 shows, not taking the all-or-nothing constraint of jobs into account hurts MIN, especially with small jobs. LIFE is 7.1× better than MIN in bin-1 and 2.5× better in bin-3 and bin-4. However, MIN's performance for bin-5 is comparable to LIFE. As Table 5 shows, the sticky policy also helps in LFU-F outperforming MIN in improving cluster efficiency.

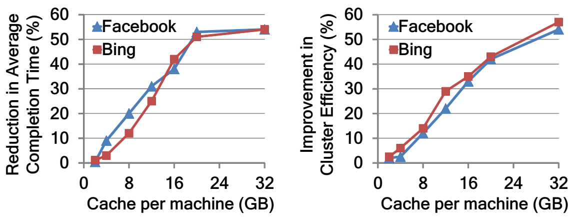
Overall, this is despite a lower cache hit-ratio. This underscores the key principle and differentiator in LIFE and LFU-F – coordinated replacement implementing the sticky policy, as opposed to simply focusing on hit-ratios.

Scheme	Facebook		Bing	
	% Job Saving	Hit Ratio (%)	% Job Saving	Hit Ratio (%)
LIFE	53%	43%	51%	39%
MIN	13%	63%	30%	68%
LRU	15%	36%	16%	34%
LFU	10%	47%	21%	48%

Table 4: **Performance of cache replacement schemes in improving average completion times. LIFE beats all its competitors despite a lower hit-ratio.**

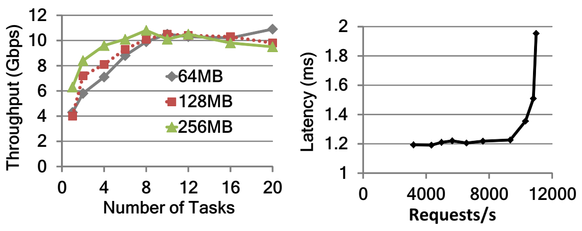
Scheme	Facebook		Bing	
	% Cluster Efficiency	Hit Ratio (%)	% Cluster Efficiency	Hit Ratio (%)
LFU-F	47%	58%	54%	62%
MIN	40%	63%	44%	68%
LRU	32%	36%	23%	34%
LFU	41%	47%	46%	48%

Table 5: **Performance of cache replacement schemes in improving cluster efficiency. LFU-F beats all its competitors despite a lower hit-ratio.**



(a) LIFE (b) LFU-F

Figure 18: **LIFE's and LFU-F's sensitivity to cache size.**



(a) PACMan Client (b) PACMan Coordinator

Figure 19: **Scalability. (a) Simultaneous tasks serviced by client, (b) Simultaneous client updates at the coordinator.**

5.5 Cache Size

We now evaluate PACMan's sensitivity to available cache size (Figure 18) by varying the budgeted cache space at each PACMan client varies between 2GB and 32GB. The encouraging observation is that both LIFE and LFU-F react gracefully to reduction in cache space. As the cache size reduces from 20GB on to 12GB, the performance of LIFE and LFU-F under both workloads hold to provide appreciable reduction of 35% in completion time and 29% improvement in cluster efficiency, respectively.

For lower cache sizes ($\leq$ 12GB), the workloads have a stronger influence on performance. While both workloads have a strong heavy-tailed distribution, recall from Table 2 that the skew between the small jobs and large jobs is higher in the Facebook workload. The high fraction of small jobs in the Facebook workload ensures that LIFE's performance drops much more slowly. Even at lower cache sizes, there are sufficient small jobs whose inputs can be retained by LIFE. Contrast with the sharper drop for caches sizes $\leq$ 12GB for the Bing workload.

LFU-F reacts more smoothly to decrease in cache space. Unlike job completion time, cluster efficiency improves even with incomplete files; the sticky policy helps improve it. The correlation between the frequency of access and size of files (§3.2), coupled with the fact that the inputs of the large jobs are bigger in the Facebook workload than the Bing workload, leads to LFU-F's performance deteriorating marginally quicker with reducing cache space in the Facebook workload.

5.6 Scalability

We now probe the scalability limits of the PACMan coordinator and client. The client's main functionality is to provide and cache blocks for tasks. We measure the throughput when tasks communicate with the client and latency when clients deal with the coordinator.

PACMan **Client:** We stress the PACMan client to understand the number of simultaneous tasks it can serve before its capacity saturates. Each task reads a block from the client's cache. Figure 19a reports the aggregate throughput for block sizes of 64MB, 128MB and 256MB. For block sizes of 128MB, we see that the client saturates at 10 tasks. Increasing the number of tasks beyond this point results in no increase in aggregate throughput. Halving the block size to 64MB only slightly nudges the saturation point to 12 tasks. We believe this is due to the overheads associated with connection management. Connections with 256MB blocks peak at 8 tasks beyond which the throughput stays constant. The set of block sizes we have tested represent the commonly used settings in many Hadoop installations. Also, since Hadoop installations rarely execute more than 8 or 10 map tasks per machine, we conclude that our client scales sufficiently to handle the expected load.

PACMan **Coordinator:** Our objective is to understand the latency added to the task because of the PACMan client's communication with the PACMan coordinator. Since we assume a single centralized coordinator, it is crucial that it supports the expected number of client requests (block updates and LIFE eviction). We vary the number of client requests directed at the server per second and observe the average latency to service those requests. As Figure 19b shows, the latency experienced by the requests stays constant at ~1.2ms until 10,300 requests per second. At this point, the coordinator's processing overhead starts increasing the latency. The latency nearly doubles at around 11,000 requests per second. Recently reported research on framework managers [10] show that the number of requests handled by the centralized job managers of Hadoop is significantly less (3,200 requests/second). Since a task makes a single request to the coordinator via the client, we believe the coordinator scales well to handle expected loads.

6 Enhancements to PACMan

We now discuss two outstanding issues with PACMan that can help improve its performance.

6.1 Optimal Replacement for Parallel Jobs

An unanswered question is the optimal cache replacement strategy to minimize average completion time of parallel jobs or maximize cluster efficiency. The optimal algorithm picks that block for replacement whose absence hurts the least when the entire trace is replayed. Note the combinatorial explosion as a greedy decision for each replacement will not be optimal. We outline the challenges in formulating such an oracular algorithm.

The completion time of a job is a function of when its tasks get scheduled, which in turn is dependent on the availability of compute slots. An aspect that decides the availability of slots for a job is its fair share. So, when executing tasks of a job finish, slots open up for its unscheduled tasks. Modeling this requires knowing the speed-up due to memory locality but that is non-trivial because it varies across tasks of even the same job. Further, scheduler policies may allow jobs to use some extra slots if available. Hence one has to consider scheduler policies on using extra slots as well as the mechanism to reclaim those extra slots (*e.g.,* killing of running tasks) when new jobs arrive. Precise modeling of the speed-ups of tasks, scheduler policies and job completion times will help formulate the optimal replacement scheme and evaluate room for improvement over LIFE and LFU-F.

6.2 Pre-fetching

A challenge for any cache is data that is accessed only once. While our workloads have only a few jobs that read such singly-accessed blocks, they nonetheless account for over 30% of all tasks. Pre-fetching can help provide memory locality for these tasks.

We consider two types of pre-fetching. First, as soon as a job is submitted, the scheduler knows its input blocks. It can inform the PACMan coordinator which can start pre-fetching parts of the input that is not in cache, especially for the later waves of tasks. This approach is helpful for jobs consisting of multiple waves of execution. This also plays nicely with LIFE's policy of favoring small single-waved jobs. The blocks whose files are absent are likely to be those of large multi-waved jobs. Any absence of their input blocks from the cache can be rectified through pre-fetching. Second, recently created data (*e.g.,* output of jobs or logs imported into the file system) can be pre-fetched into memory as they are likely to be accessed in the near future, for example, when there are a chain of jobs. We believe that an investigation and application of different pre-fetching techniques will further improve cluster efficiency.

7 Related Work

There has been a humbling amount of work on in-memory storage and caches. While our work borrows and builds up on ideas from prior work, the key differences arise from dealing with parallel jobs that led to a coordinated system that improved job completion time and cluster efficiency, as opposed to hit-ratio.

RAMCloud [18] and prior work on databases such as MMDB [16] propose storing all data in RAM. While this is suited for web servers, it is unlikely to work in data-intensive clusters due to capacity reasons – Facebook has 600× more storage on disk than aggregate memory. Our work thus treats memory as a constrained cache.

Global memory systems such as the GMS project [5], NOW project [6] and others [14] use the memory of a remote machine instead of spilling to disk. Based on the vast difference between local memory and network throughputs, PACMan's memory caches only serves tasks on the local node. However, nothing in the design precludes adding a global memory view. Crucially, PACMan considers job access patterns for replacement.

Web caches have identified the difference between byte hit-ratios and request hit-ratios, i.e., the value of having an entire file cached to satisfy a request [9, 12, 23]. Request hit-ratios are best optimized by retaining small files [26], a notion we borrow. We build up on it by addressing the added challenges in data-intensive clusters. Our distributed setting, unlike web caches, necessitate coordinated replacement. Also, we identify benefits for partial cache hits, *e.g.,* large jobs that benefit with partial memory locality. This leads to more careful replacement like evicting parts of an incomplete file. The analogy with web caches would not be a web request but a web *page* – collection of multiple web objects (.gif, .html). Web caches, to the best of our knowledge, have not considered cache replacment to optimize at that level.

LIFE's policy is analogous to servicing small requests in queuing systems, *e.g.,* web servers [19]. In particular, when the workload is heavy-tailed, giving preference to small requests hardly hurts the big requests.

Distributed filesystems such as Zebra [17] and xFS [8] developed for the Sprite operating system [22] make use of client-side in-memory block caching, also suggesting using the cache only for small files. However, these systems make use of relatively simple eviction policies and do not coordinate scheduling with locality since they were designed for usage by a network of workstations.

Cluster computing frameworks such as Piccolo [25] and Spark [21] are optimized for iterative machine learning workloads. They cache data in memory after the first iteration, speeding up further iterations. The key difference with PACMan is that since we assume no application semantics, our cache can benefit multiple and a greater variety of jobs. We operate at the storage level and can serve as a substrate for such frameworks to build upon.

8 Conclusion

We have described PACMan, an in-memory coordinated caching system for data-intensive parallel jobs. Parallel jobs run multiple tasks simultaneously in a wave, and have the all-or-nothing property, *i.e.,* a job is sped up only when inputs of all such parallel tasks are cached. By globally coordinating access to the distributed caches, PACMan ensures that a job's different tasks, distributed across machines, obtain memory locality. On top of its coordinated infrastructure, PACMan implements two cache replacement policies – LIFE and LFU-F – that are designed to minimize average completion time of jobs and maximize efficiency of the cluster. We have evaluated PACMan using a deployment on EC2 using production workloads from Facebook and Microsoft Bing, along with extensive trace-driven simulations. PACMan reduces job completion times by 53% and 51% (small interactive jobs improve by 77%), and improves efficiency by 47% and 54%, respectively. LIFE and LFU-F outperform traditional replacement schemes, including MIN.

Acknowledgments

We thank Facebook and Microsoft Bing for access to job traces from their production clusters. We are also grateful to our shepherd, Eddie Kohler, and the anonymous reviewers whose inputs greatly improved the paper. For feedback on the draft, we thank Arka Bhattacharya, Mosharaf Chowdhury, Tathagata Das, Aurojit Panda, Ariel Rabkin and David Zats of the AMP Lab at Berkeley, and Sriram Rao of Yahoo! Research.

References

[1] Amazon Elastic Compute Cloud. `http://aws.amazon.com/ec2/instance-types/`.

[2] Hadoop Distributed File System. `http://hadoop.apache.org/hdfs`.

[3] Hadoop Slowstart. `https://issues.apache.org/jira/browse/MAPREDUCE-1184/`.

[4] Hive. `http://wiki.apache.org/hadoop/Hive`.

[5] The Global Memory System (GMS) Project. `http://www.cs.washington.edu/homes/levy/gms/`.

[6] The NOW Project. `http://now.cs.berkeley.edu/`.

[7] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, E. Harris, and B. Saha. Reining in the Outliers in Map-Reduce Clusters using Mantri. In *USENIX OSDI*, 2010.

[8] T. E. Anderson, M. D. Dahlin, J. M. Neefe, D. A. Patterson, D. S. Roselli, and R. Y. Wang. Serverless Network File Systems. In *ACM SOSP*, 1995.

[9] M. Arlitt, L. Cherkasova, J. Dilley, R. Friedrich, and T. Jin. Evaluating Content Management Techniques for Web Proxy Caches. In *WISP*, 1999.

[10] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. Joseph, R. Katz, S. Shenker, I. Stoica. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. In *USENIX NSDI*, 2011.

[11] Laszlo A. Belady. A Study of Replacement Algorithms for Virtual-Storage Computer. *IBM Systems Journal*, 1966.

[12] L. Cherkasova and G. Ciardo. Role of Aging, Frequency, and Size in Web Cache Replacement Policies. In *HPCN Europe*, 2001.

[13] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *USENIX OSDI*, 2004.

[14] M. J. Franklin, M. J. Carey, and M. Livny. Global Memory Management in Client-Server Database Architectures. In *VLDB*, 1992.

[15] G. Ananthanarayanan, S. Agarwal, S. Kandula, A. Greenberg, I. Stoica, D. Harlan, E. Harris. Scarlett: Coping with Skewed Popularity Content in MapReduce Clusters. In *ACM EuroSys*, 2011.

[16] H. Garcia-Molina and K. Salem. Main Memory Database Systems: An Overview. In *IEEE Transactions on Knowledge and Data Engineering*, 1992.

[17] John H. Hartman and John K. Ousterhout. The Zebra Striped Network File System. In *ACM SOSP*, 1993.

[18] J. Ousterhout *et al.* The Case for RAMClouds: Scalable High-Performance Storage Entirely in DRAM. In *SIGOPS Operating Systems Review*, 2009.

[19] M. Harchol-Balter M. E. Crovella, R. Frangioso. Connection Scheduling in Web Servers. In *USENIX USITS*, 1999.

[20] M. Isard, M. Budiu, Y. Yu, A. Birrell and D. Fetterly. Dryad: Distributed Data-parallel Programs from Sequential Building Blocks. In *ACM Eurosys*, 2007.

[21] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, I. Stoica. Spark: Cluster Computing with Working Sets. In *USENIX HotCloud*, 2010.

[22] M. N. Nelson, B. B. Welch, and J. K. Ousterhout. Caching in the Sprite Network File System. *ACM TOCS*, Feb 1988.

[23] P.Cao and S.Irani. Cost Aware WWW Proxy Caching Algorithms. In *USENIX USITS*, 1997.

[24] R. Chaiken, B. Jenkins, P. Larson, B. Ramsey, D. Shakib, S. Weaver, J. Zhou. SCOPE: Easy and Efficient Parallel Processing of Massive Datasets. In *VLDB*, 2008.

[25] R. Power and J. Li. Piccolo: Building Fast, Distributed Programs with Partitioned Tables. In *USENIX OSDI*, 2010.

[26] S. Williams, M. Abrams, C. R. Standridge, G. Abdulla, and E. A. Fox. Removal Policies in Network Caches for World-Wide Web Documents. In *ACM SIGCOMM*, 1996.

Re-optimizing Data-Parallel Computing

Sameer Agarwal[1,3], Srikanth Kandula[1], Nicolas Bruno[2], Ming-Chuan Wu[2], Ion Stoica[3], Jingren Zhou[2]
[1]*Microsoft Research,* [2]*Microsoft Bing,* [3]*University of California, Berkeley*

Abstract– Performant execution of data-parallel jobs needs good execution plans. Certain properties of the code, the data, and the interaction between them are crucial to generate these plans. Yet, these properties are difficult to estimate due to the highly distributed nature of these frameworks, the freedom that allows users to specify arbitrary code as operations on the data, and since jobs in modern clusters have evolved beyond single map and reduce phases to logical graphs of operations. Using fixed apriori estimates of these properties to choose execution plans, as modern systems do, leads to poor performance in several instances. We present RoPE, a first step towards re-optimizing data-parallel jobs. RoPE collects certain code and data properties by piggybacking on job execution. It adapts execution plans by feeding these properties to a query optimizer. We show how this improves the future invocations of the same (and similar) jobs and characterize the scenarios of benefit. Experiments on Bing's production clusters show up to 2× improvement across response time for production jobs at the 75th percentile while using 1.5× fewer resources.

1. INTRODUCTION

In most production clusters, a majority of data parallel jobs are logical graphs of map, reduce, join and other operations [5, 6, 21, 24].

An execution plan represents a blueprint for the distributed execution of the job. It encodes, among other things, the sequence in which operations are to be done, the columns to partition data on, the degree of parallelism and the implementations to use for each operation.

While much prior work focuses on executing a given plan well, such as dealing with stragglers at runtime [1], placing tasks [15, 25] and sharing the network [8, 23], little has been done in choosing appropriate execution plans.

Execution plan choice can alleviate some runtime concerns. For example, outliers are less bothersome if even the most expensive operation is given enough parallelism.

But plan choice can do much more– it can avoid needless work (for example, by deferring expensive operations till after simpler or more selective operations) and it can trade-off one resource type for another to speed up jobs (for example, in certain cases, some extra network traffic can avoid a read/write pass on the entire data set). However, plan choice is more challenging because the space of potential plans is large and also because the appropriateness of the plan depends on the interplay between code, data and cluster hardware.

Early data-parallel systems force developers to specify the execution plan (e.g., Hadoop). To shield developers from coping with these details, some recent proposals raise the level of abstraction. A few use hand-crafted rules to generate execution plans (e.g., HiveQL [24]) or use compiler techniques (e.g., FlumeJava [6]). Other declarative frameworks cast the execution plan choice as the traditional query optimization problem (e.g., SCOPE [5], Tenzing [7], Pig [21]).

A central theme, across all schemes, is the absence of insight into certain properties of the code (such as expected CPU and memory usage), the data (such as the frequency of key values), and the interaction between them (such as the selectivity of an operation). These properties crucially impact the choice of execution plans.

Our experience with Bing's production clusters shows that these code and data properties vary widely. Hence, using fixed apriori estimates leads to performance inefficiency. Even worse, not knowing these properties constrains plan choice to be pessimistic; techniques that provide gains in certain cases but not all cannot be used.

This paper presents RoPE[1], a first step towards re-optimizing data parallel jobs, i.e., adapting execution plans based on estimates of code and data properties. To our knowledge, we are the first to do so. The new domain brings challenges and opportunities. Accurately estimating code and data properties is hard in a distributed context. Predicting these properties by collecting statistics on the raw data stored in the file-system is not practical due to the prevalence of user-defined operations. But, knowing these properties enables a large space of improvements that is disjoint from prior work and so are the methods to achieve these improvements.

The sheer number of jobs indicates that the estimation and use of properties has to be automatic. RoPE piggybacks estimators with job execution. The scale of the data and distributed nature of computation means that no single task can examine all the data. Hence, RoPE collects statistics at many locations and uses novel ways to compose them. To keep overheads small, RoPE's collectors can only keep a small amount of state and work in a single pass over data. Collecting meaningful properties, such as the number of distinct values or heavy hitters, under these

[1]*R*eoptimizer for *P*arallel *E*xecutions

constraints precludes traditional data structures and leads to some interesting designs.

The flexibility allowed for users to define arbitrary code leads to a much tighter coupling between data and computation in data parallel clusters. As long as they conform to well-defined interfaces, users can submit jobs with binary implementations of operations. In this context, predicting code properties becomes even more difficult. Traditional database techniques can project statistics on the raw data past some simple operations with alphanumeric expressions but doing so through multiple operations, more complex expressions, potentially dependent columns and user-defined operations introduces impractically large error [3]. Rather than predicting, RoPE instruments job execution to measure properties directly.

We find traditional work on adaptive query optimization to be specific to the environment of one database server [2, 3, 16] and the resulting space of optimizations. For example, a target scenario minimizes the reads from disk by keeping one side of the join in the server's memory. RoPE translates these ideas to the context of distributed systems and parallel plans. In doing so, RoPE uses a few aspects of the distributed environment that make it a better fit for adaptive optimization. Unlike the case of a database server where most queries finish quickly and the server has to decide whether to use its constrained resources to run the query or to re-optimize it, map-reduce jobs last much longer and the resources to re-optimize are only a small fraction of those used by the job. Further, if a better plan becomes available, transitioning from the current plan to the better plan is tricky in databases [16] whereas data parallel jobs have many inherent barriers at which execution plans can be switched.

In Bing's production clusters, we observe that many key jobs are re-run periodically to process newly arriving data. Such recurring jobs contribute 40.32% of all jobs, 39.71% of all cluster hours and 26.07% of the intermediate data produced. We also observe that the code and data properties are remarkably stable across recurring jobs despite the fact that each job processes new data. These jobs are RoPE's primary use-case.

RoPE adapts the execution plans for future invocations of such jobs by feeding the observed properties into a cost-based query optimizer (QO). Our prototype is built atop SCOPE [5], the default engine for all of Bing's map-reduce clusters, but our techniques can be applied to other systems. The optimizer evaluates various alternatives and chooses the plan with the least expected job latency. Additionally, we modified the optimizer to use the actual code and data properties while estimating costs. Working with a QO enables RoPE to not only perform local changes such as changing the degree of parallelism of operations, but also changes that require a global context, such as reordering operations, choosing appropriate operation implementations and grouping operations that have little work to do into a single physical task.

We find jobs that are not completely identical often have common *parts*. Further, during the execution of a job, while some of the global changes are logically impossible due to operations that have already executed, other changes remain feasible. RoPE can help in both these cases since (a) the query optimizer can work with incomplete estimates and (b) the code and data properties are linked to the sub-graph at which they were collected and can be matched to other jobs with an identical sub-graph.

Our contributions include:

- Based on experiences and measurements on Bing's production clusters, we describe scenarios where knowledge of code and data properties can improve performance of data-parallel jobs. A few of these scenarios are novel (§2).
- Design of the first re-optimizer for data-parallel clusters, which involves collecting statistics in a distributed context, matching statistics across sub-graphs and adapting execution plans by interfacing with a query optimizer (§3).
- Results from a partial prototype, deployed on production clusters, which show RoPE to be effective at reoptimizing jobs (§4, §5). Production jobs speed up by over 2× at the 75th percentile while using 1.5× fewer resources. RoPE achieves these gains by designing better execution plans that avoid wasteful work (reads, writes, network shuffles) and balance operations that run in parallel.

A user would expect her data-parallel jobs to run quickly. She would expect this even though the code is unknown, even though the data properties are hard to estimate, even though the code and data interact in unpredictable ways, and even though the code, the data and the cluster hardware and software keep evolving. RoPE is a first step towards reoptimizing data-parallel computing.

2. COST OF IGNORING CONTEXT

It is not uncommon for data-parallel computing frameworks such as Dryad and MapReduce to process petabytes of data each day. However, their inability to leverage data and computation statistics renders them unable to generate execution plans that are better suited for the jobs they run and prevents them from utilizing historical context to improve future executions. Here, we describe axiomatic scenarios of such inefficiency and quantify both their impact and frequency of occurrence.

2.1 Background

Our experience is rooted in Bing's production clusters consisting of thousands of multi-core servers participating in a distributed file system that supports

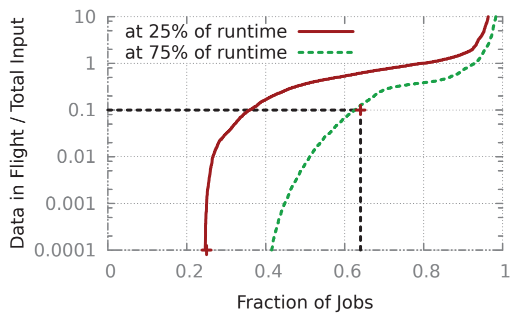


Figure 1: Data that remains in flight when a job has executed for 25% (or 75%) of its running time.

both structured and unstructured data. Jobs are written in SCOPE [5], a SQL-like mashup language with support for arbitrary user-defined operators. That is, users specify their data parallel jobs within a declarative framework (e.g., `select, join, group by`) but are allowed to declare their own implementations of operators as long as they fit the templates provided (e.g., `extractor, processor, combiner, reducer`). A compiler translates the query into an execution plan which is then executed in parallel on a Dryad-like [14] runtime engine. Plans are directed acyclic graphs where edges represent dataflow and nodes represent work that can be executed in parallel by many tasks. A task can consist of multiple operations. A job manager orchestrates the execution of the job's plan by issuing tasks when their inputs are ready, choosing where tasks run on the cluster, and reacting to outliers and failures. To facilitate better resource allocation across concurrent jobs, individual job managers work in close contact with a per-cluster global manager.

Unless otherwise specified, our results here use a dataset that contains all the events from a large production cluster in Bing. The events encode for each entity (job/ operation/ task/ or network transfer), the start and end times of the entity, the resources used, the completion status, and its dependencies with other entities. Our experiments are based on examining all events during the month of September 2011 on a cluster consisting of tens of thousands of servers.

2.2 Little Data

We notice that while most map-reduce programs start off by reading a large amount of data, each successive operation (filters, reduces, etc.) produces considerably fewer output compared to its input. Hence, more often than not, just after a small number of these consecutive operations there is very little data left to process. We call this the *little data* case. The little-data observation can be used to optimize the tail of most jobs. In some cases, the degree of parallelism on many operations in the tail can be reduced. This saves scheduling overhead on the un-necessary tasks. In other cases, multiple operations in the tail can be coalesced into a single physical operation, i.e., one group of tasks executes these operations in parallel. This avoids needless checkpoints to disk. In yet other cases, broadcast joins can be used instead of pair-wise joins (see §2.6) thereby saving on network shuffle and disk accesses. The challenge however is that the reduction factors are unknown apriori and vary by several orders of magnitude.

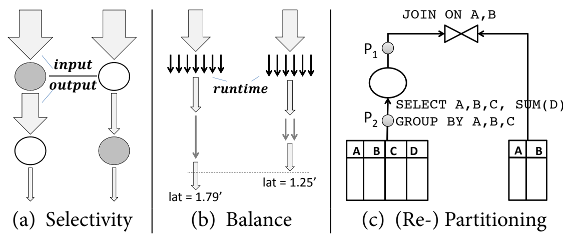
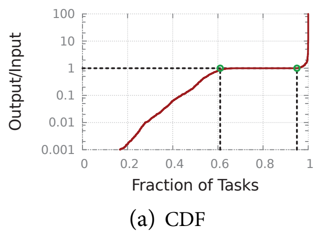


Figure 2: Motivating examples for re-optimizing data parallel computing



Range	Fraction
0	3.7%
(0, .001)	12.9%
[.001, .1)	27.2%
[.1, .5)	11.3%
[.5, .9)	6.7%
[.9, 1]	32.8%
> 1	5.4%
> 2	.35%
> 10	.01%

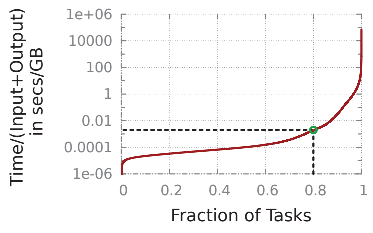
(b) readout

Figure 3: Variation in selectivity across tasks

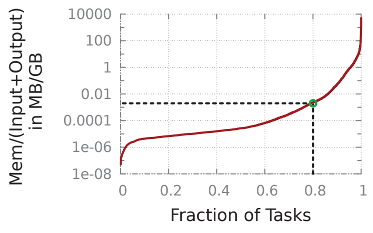
Fig. 1 plots the fraction of input data that remains *in flight* after jobs have been running for 25% (and 75%) of their runtime. We compute the data in-flight at any time by taking a cut of the job's execution graph at that time and adding up the data exchanged between tasks that are on either side of this cut. For convenience, we place tasks running at that time to the left of the cut. We see that while some jobs have more data in flight than their input (above $y = 1$ line), most of the jobs have much fewer. In fact for over 20% of jobs, the data in flight reduces to less than $\frac{1}{10^4}$ of their input within a quarter of the job's running time and over 60% of jobs have less than a tenth of data in flight after three quarters of their running time. This means that little data, and the above optimizations, can be brought to bear.

2.3 Varying Selectivity, Reordering

Consider a pair of commutative operations. Ordering them, so that the more selective operation (one with a lower output to input ratio) runs first will *avoid work* thereby saving compute hours, disk accesses and network shuffle. See Fig. 2(a), where the width of block arrows represent the data flow between a pair of commutative operators (indicated by the circles). Evidently, the plan on the right avoids processing unnecessary data and potentially saves significant cluster cycles by appropriately ordering the operators based on their selectivity. These pairs hap-



(a) Runtime/Data Processed



(b) Mem Usage/ Data Processed

Figure 4: Variation in Operation costs; in time to process and memory usage, per unit data processed

pen often, due to operations that are independent of each other (e.g. operations on different columns) or are commutative. Identifying these pairs can be hard in general but SCOPE's declarative syntax allows the use of standard database techniques to discover such pairs. Finding the selectivities of operations remains a challenge.

Standard database techniques to predict operator selectivity are hard to translate to map-reduce like frameworks due to the complexity of expressions and long sequences of operations. The selectivity of alphanumeric expressions (e.g. `select on X=30`) can be predicted by using clever histograms on the raw data (e.g. equi-depth) but creating these histograms requires many passes over the data. Predicting the selectivity for user-defined operations (e.g. `select when columnvalue.BeginsWith("http://bing")`) is an open problem [3]. We see such code in a majority of jobs. Moreover, the prediction errors grow exponentially with the length of the sequence of operations [3]. Computing more detailed synopsis on a random sample is often of only marginal benefit [18]. Finally, correlations between sets of columns, as is common, increases prediction error (e.g. `select on X=30 and Y=10` can produce just as much data as the `select on X=30` or much less). RoPE estimates selectivity by direct instrumentation.

Fig. 3(a) plots a CDF of the selectivity (ratio of output to input) of operations in our dataset. Note the y axis is in log scale. About 5% of operations produce more data than they consume (above $y = 1$). These are typically (outer) joins. About 34% have output roughly equalling input. The remaining 60% operations produce less output than their input but the selectivity varies widely– 17% produce fewer than $\frac{1}{1000}$'th of their input, and the coefficient of variation ($\frac{stdev}{mean}$) is 1.3 with a range from 0 to 171. This means that if these selectivities were available, there is substantial room to reorder operators.

2.4 Varying Costs, Balance

Suppose we figured out selectivities and picked the right order. Uniquely for data parallel computing, we need to choose the number of parallel instances for each operation. Choosing too few instances will cause that operation to become a bottleneck as per Amdahl's law. On the other hand, choosing too many leads to needless per-task scheduling, queuing and other startup overhead. *Balance*, i.e., ensuring that each operation has enough parallelism and takes roughly the same amount of time, can improve performance significantly [22]. See Fig. 2(b) where block arrows again represent the dataflow and the thin arrows now represent the tasks in each of the two operations. Here, using one less task for the upstream operation and one more for the downstream operation reduces job latency by over 30%.

Achieving balance in the context of general data parallel computing is hard because the costs (runtime, memory etc.) of the operators are unknown apriori. These costs depend on the amounts of data processed, the types of computation performed and also on the type of data. For example, a complex sentence can take longer to translate than a simpler one of the same length. Even worse, *late-binding*, i.e., deferring the choice of the amount of parallelism to the runtime is hard because local changes have global impact; for example, the number of partitions output by the map phase restricts the maximum number of reduce tasks at the next stage. RoPE estimates these costs to generate balanced execution plans.

Fig. 4(a) plots a CDF of the runtime per unit byte read or written by operations in the dataset. The analogous plot for memory used by tasks is in Fig. 4(b). Note again that the y axes are in log scale. While as variable as their selectivity– the middle 50^{th} percent of operators have costs spread over two orders of magnitude– we find that operator costs skew more towards higher values. Per unit data processed, 20% of operations take over 100X more time and memory than the average over the remaining operations. It is crucial to identify these heavy operations, to offset their costs by increasing the parallelism and for the case of memory, to place tasks so that they do not compete with other memory hungry tasks.

A consequence of unpredictable data selectivity and operator costs is the lack of balance. We find that our compiler both underestimates and overestimates an operation's work. Fig. 5 plots a CDF of the runtime of the median task in each stage. The y axis is in log scale. The median task in a stage is unlikely to be impacted by failures or be an outlier. So, if the compiler apportions parallelism well, the median task in each stage should take

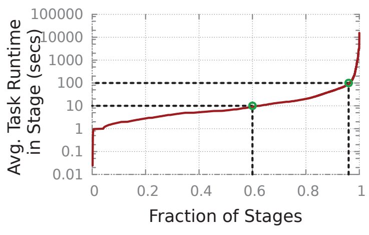


Figure 5: Imbalance

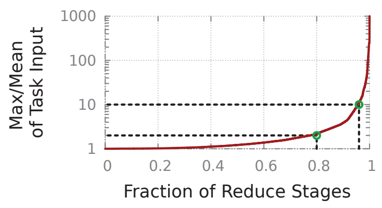


Figure 6: Skewness

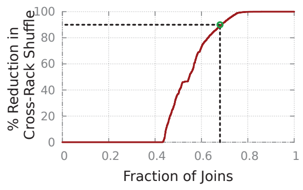


Figure 7: Replacing Pair-wise Joins with Broadcast Joins

about the same time. This duration could be chosen to trade-off fault-tolerance vs. the cost of checkpointing to disk. However, we see that while roughly 60% of all tasks finish within 10 seconds, 4% take over 100s with the last 1% taking over 1000s. We found the tail dominated by tasks with user defined operators. Setup and scheduling overheads outweigh the useful work in short-lived tasks whereas the long-lived tasks are bottlenecks in the job.

2.5 Partition Skew and Repartitioning

A key to efficient data-parallel computing is to avoid skews in partition and to re-partition only when needed. Consider the example in Fig. 2(c). The naive implementation would partition the data twice– once on A, B, C (at P_1), followed by a network shuffle and a reduce to compute the sum, and then again on A, B (at P_2) followed by another shuffle for the join. It is tempting to just partition the data once, say on A, B to avoid the network shuffle and the pass over data. Note that partitioning on fewer keys does not violate the correctness of the reduce that computes SUM(D). Each reduce task will now compute many rows, one per distinct value of C, rather than just the one row they would have produced were the map to partition on all three columns. However, if there is not enough entropy on A and B, i.e., only a few distinct values have many records, then partitioning on the sub-group can make things worse. A few reduce tasks might receive a lot of data to process while other reduce tasks have none, and the overall parallel execution can slow down.

Fig. 6 estimates how skewed the partitions can be in our cluster. Note that our compiler is conservative and does not partition on sub-groups to avoid re-partitioning. Yet significant skew happens. We define skew as the ratio of the maximum data processed by a reduce task to the average over other tasks in that reduce phase. We see that in 20% of the reduce stages, the largest partition is twice as large as the average and in about 5% of the stages the largest partition is over ten times larger than the average. Such skew causes unequal division of work and bottlenecks but can be avoided if the data properties are known.

2.6 From operations to implementations

Often, the same operation can be implemented in several ways. While choosing the appropriate implementation can result in significant improvements, doing so requires appropriate context that is not available in today's

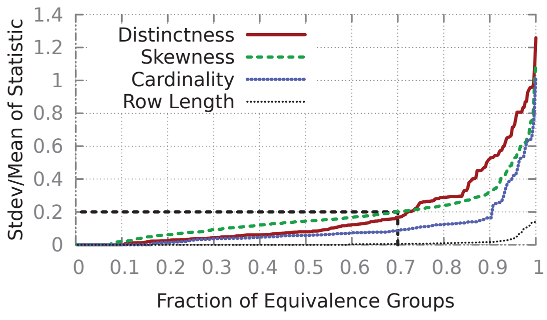


Figure 8: Stability of data properties across recurring jobs

systems. For example, consider `Join`. The default implementation `PairJoin`, involves a map-reduce operation on each side that partitions data on the join columns. This causes three read/write passes on each of the sides and at least one shuffle each across the network. However, if one of the sides is smaller, perhaps due to the *little data* case (§2.2), one could avoid shuffling the larger side and complete the join in one pass on that side. The trick is to broadcast the smaller side to each of the tasks that is operating in parallel on the larger side. The problem though is that when used inappropriately, a `BroadcastJoin` can be even more expensive than a `PairJoin`.

Fig. 7 plots the potential benefits of replacing `PairJoins` with `BroadcastJoins`. It shows the amount of data shuffled in either case. We see that about 40% of joins in the dataset would see no benefit. This can happen if both join inputs are considerably large and/or when the parallelism on the larger side is so much that broadcasting the smaller input dataset to too many locations becomes a bottleneck. However, off the remaining joins, the median join shuffles 90% less data when using broadcast joins.

2.7 Recurring Jobs

We find that many jobs in the examined cluster repeat and are re-run periodically to execute on the newly arriving data. Such recurring jobs contribute to 40.32% of all jobs, 39.71% of all cluster hours and 26.07% of intermediate data produced. If the extracted statistics are stable per recurring job, i.e., the operations behave statistically similar to the previous execution when running on newer data of the same stream, then RoPE's instrumentation would suffice to re-optimize future invocations.

Fig. 8 plots the average difference between statistics collected at the same location in the execution plan across recurring jobs. We picked all of the recurring jobs from one business group and instrumented five different runs of each job. While most of these jobs repeated daily, a few repeated more frequently. The figure has four distributions, one per data property that RoPE measures. We defer details on the specifics of the properties (see Table 1, §3.1) but note that while some properties, such as row length, are more predictable than others, the overall statistics are similar across jobs– the ratio $\frac{stdev}{mean}$ is less than 0.2 for 70% of the locations.

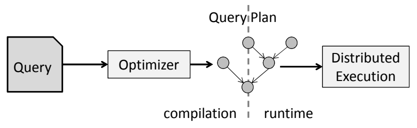
2.8 Current Approaches, Alternatives

To the best of our knowledge, we are the first to re-optimize data parallel jobs by leveraging data and computation statistics. Current frameworks use best-guess estimates on the selectivity and costs of operations. Such rules-of-thumb, as we saw in §2.2–§2.6, perform poorly. Hadoop and the public description of MapReduce leave the choice of execution plans, the number of tasks per machine and even low-level system parameters such as buffer sizes to the purview of the developer. HiveQL [24] uses a rule-based optimizer to translate scripts written in a SQL-like language to sequences of map-reduce operations. Starfish [12] provides guidance on system parameters for Hadoop jobs. To do so, it builds a machine learning classifier that projects from the multi-dimensional parameter space to expected performance but does not explore semantic changes such as reordering. Ke et. al. [17] propose choosing the number of partitions based on operator costs. In contrast, RoPE can perform more significant changes to the execution plan, similar to FlumeJava [6] and Pig [21], but additionally does so based on actual properties of the code and data.

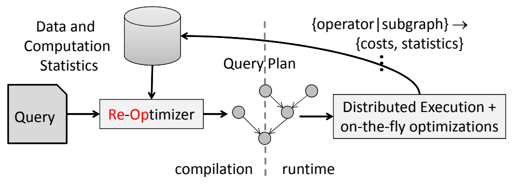
It is tempting to ask end-users to specify the necessary context, for e.g., tag operations with cost and selectivity estimates. We found this to be of limited use for a few reasons. First, considerable expertise and time is required to hand-tune each query. Users often miss opportunities to improve or make mistakes. Second, exposing important knobs to a wide set of users is risky. Unknowingly, or greedily, a user can hog the cluster and deny service to others; for instance, by tricking the system to give her job a high degree of parallelism. Finally, changes in the script, the cluster characteristics, the resources available at runtime or the nature of data being processed can require re-tuning the plan. Hence, RoPE re-optimizes execution plans by automatically inferring these statistics.

2.9 Experience Summary, Takeaways

Note that all of the problems described here happen deterministically. They are not due to heterogeneity or runtime variations [1] or due to poor placement of tasks [15, 25] or due to sharing the cluster [13, 23]. We believe that



(a) Current Cluster(s)


Data and Computation Statistics
{operator|subgraph} → {costs, statistics}
Query Plan
Query
Re-Optimizer
Distributed Execution + on-the-fly optimizations
compilation
runtime

(b) With RoPE

Figure 9: RoPE's architecture for re-optimization

the choice of execution plan is orthogonal to these problems that arise *during* the execution of a plan. The underlying cause is that predicting relevant data and computation statistics deep into a job's execution is necessary to find a good plan but is difficult due to the intricate coupling between data and computation.

When employed together, these improvements add up to more than their sum. Promoting a more selective operator closer to the input, can reduce the data flowing in so much that a subsequent join may be implemented with broadcast join. We found in practice that such global changes to the execution plan accrue more benefits than making singleton changes. RoPE achieves both types of re-optimizations as we will see in §3.

A few key takeaways follow.

- Being un-aware of data and computation context leads to slower responses and wasted resources.
- Unlike the case of singleton database servers, data-parallel computation provides different space for improvements and has new challenges such as coping with arbitrary user defined operations and expressions.
- Global changes to the execution plan add more value than local ones.

3. DESIGN

RoPE enables re-optimization of data parallel computing. To obtain data- and computation- context, RoPE interposes instrumentation code into the job's dataflow. An operation can be instrumented with collectors on its input(s), output(s) or both. We describe how RoPE chooses what information to collect, avoids redundancy in the locations from which stats are collected, and the algorithms to compose statistics from distributed locations in §3.1. These statistics are funneled to the job manager and are used to improve execution plans in a few different ways, each differing in the scope of possible changes and the complexity to achieve those changes.

Even though an execution plan is already chosen for a running job, RoPE uses the statistics collected during the run to improve some aspects of the job. Descendant stages that are at least a barrier away from the stages where datastats are being collected will not have begun execution. The implementation of these stages can be changed on the fly. Stages that are pipelined with the currently executing stage can be changed, since inter-task data flow happens through the disk. Some plan changes may be constrained by stages (or parts of stages) that have already run. Hence RoPE performs changes that only impact the un-executed parts of the plan, such as altering the degree of parallelism of non-reduce stages.

RoPE uses the collected statistics to generate better execution plans for new jobs. Here, RoPE can perform more comprehensive changes. Recall from §2.7 that many jobs in the examined cluster recur because they periodically execute on new data and that the extracted datastats are stable across runs of these jobs. In this case, upon a new run of a recurrent job, datastats collected from previous runs are used as additional inputs to the plan optimizer. We describe how statistics are stored so they can be matched with expressions from subsequent jobs in §3.2. We also note that partial overlaps are common among jobs [11] and our matching framework extends to cover this case. The methodology of how the statistics are used is described in §3.3. An illustrative case study of the changes that RoPE achieves is in §5.2.

3.1 Collecting contextual information

Choosing what to observe and the statistics to collect has to be done with care since the context we collect will determine the improvements that we can make. Broadly, we collect statistics about the resource usage (e.g., CPU, memory) and data properties (e.g., cardinality, number of distinct values). See Table 1 for a summary.

The nature of map-reduce jobs leads to a few unique challenges. First, map-reduce jobs examine a lot of data in a distributed fashion. There is no instrumentation point that observes all the data and even if created, such instrumentation would not scale with the data size. Hence, we require stat collection mechanisms to be *composable*, i.e., local statistics obtained by looking at parts of the data should be composable into a global statistic over all data. Second, to be applicable to a wide set of jobs, the collection method should have *low overhead*, i.e., overhead that is only a small fraction of the task that it piggybacks upon. In particular, the memory used should scale sub-linearly with data size and to limit computation cost, the statistics should be collected in a single pass. Together, these constraints are quite strict, so we adapt some pre-existing streaming and approximation algorithms.

Finally, to be useful, the statistics have to be *unambiguous*, *precise*, and have *high coverage*. By unambiguous, we mean that the statistics should contain metadata describing the location in the query tree that these

Type	Description	Granularity
Data Properties	Cardinality	Query Subgraph
	Avg. Row Length	Query Subgraph
	# of Distinct values	Column @ Subgraph
	Heavy hitter values, their frequency	Column @ Subgraph
Code Properties	CPU and Memory Used per Data read and written	Task
Leading Statistics	Hash histogram	Column @ Subgraph
	Exact sample	Column @ Subgraph

Table 1: Statistics that RoPE collects for reoptimization

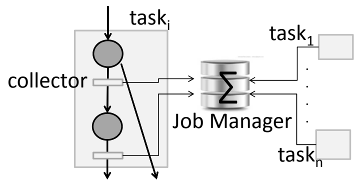


Figure 10: A task can have many operations and hence, collectors. A job manager composes individual statistics.

statistics were collected at. Global changes to the plan, during re-optimization for subsequent jobs for example, can alter the plan so much that previously observed subtrees no longer occur. Precision is an accuracy metric; we try to match the accuracy requirements of the improvements that we would like to make with the properties of the algorithmic techniques that we use to collect statistics. For coverage, we would like to observe as many different points in the job execution as possible. However, to keep costs low, we ignore instrumenting operations whose impact on the data is predictable (e.g., the input size of a sort operation is the same as its output). Further, we only look at the interesting columns. That is, we collect column-specific statistics only on columns whose values will, at some later point in the job, be used in expressions (e.g. record stats for `col` if `select col=...`, `join by col=...`, or `reduce on col` follow).

Implementation: RoPE interposes stat-collectors at key points in the job's dataflow. Datastat collectors are pass-through operators which keep negligible amounts of state and add little overhead. We also extend the task wrapper to collect the resources used by the task. When a task finishes, all datastats are ferried to the job manager (see Figure 10), which then composes the stats. The stats are used right away and also stored with a matching service for use with future jobs (see Figure 9(b)).

3.1.1 Data Properties

At each collection point we collect the number of rows and the average row length. This statistic will inform whether data grows or shrinks and by how much as it flows through the query tree. Composing these statistics is easy–for example, the total number of rows after a `select` operation is simply the sum of the number of rows seen by the collectors that observe the output of that `select` across all the tasks that contain that `select`.

Further, for each interesting column, we compute the number of distinct values and the heavy hitters, i.e., values that repeat in a large fraction of the rows. These statistics, as we will see shortly, help avoid skews during partition and also help pick better implementations. Computing these statistics while keeping only a small amount of state in a single pass is challenging, let alone the need to compose across different collection points.

Our solution builds on some state-of-the-art techniques that we carefully chose because we could extend them to be composable. We will only sketch the basic algorithms and focus on how we extended them.

Lossy Counting to find Heavy Hitters.
Suppose we want to identify all values that repeat more frequently than a threshold, say 1% of the data size N. Doing this in one pass, naively, requires tracking running counts of all distinct values and uses up to $O(N)$ space.

Lossy counting [19] is an approximate streaming algorithm, with parameters s, ε that has these properties. First, it guarantees that all values with frequency over sN will be output. Further no value with a frequency smaller than $(s-\varepsilon)N$ will be output. Second, the worst case space required to do so is (sub-linear) $\frac{1}{\varepsilon}\log(\varepsilon N)$. In practice, we find that the usage is often much smaller. Third, the frequency estimated undercounts the true frequency of the elements by at most εN. The key technique is rather elegant; it tracks running frequency counts but after every $\lceil \frac{1}{\varepsilon} \rceil$ records, it retires values that do not pass a test on their frequency. For more details, please refer [19].

RoPE uses a distributed form of lossy counting. Each stat collector employs lossy counting on the subset of data that their task observes with parameters $s = 2\varepsilon, \varepsilon$. To compute heavy hitters over all the data, we add up the frequency estimates over all collectors and report distinct values with count greater than εN. Interestingly, composing in this manner retains the properties of lossy counting with slight mods. A proof sketch follows.
Proof Sketch: Let N_i be the number of records observed by the i'th collector. Note that $s-\varepsilon = \varepsilon$ and $\sum N_i = N$ First, for the frequency estimation error, if a value is reported by stat collector i, we know that its frequency estimate is no worse off than εN_i. If the value is not reported, its frequency estimate is zero; but by the existence constraint, we know that the element did not occur more than $2\varepsilon N_i$ times at this collector. Summing up the errors across collectors, we conclude that the global estimate is not off the true frequency by more than $2\varepsilon N$. Second, for false positives, we note that we only keep values whose cumulative recorded count is greater than εN, that means their true frequency is at least εN. Third, for false negatives, suppose a value has a global frequency greater than $f > 3\varepsilon N$, then it has to occur more than $3\varepsilon N_i > sN_i$ times at some collector, and so will be reported. Even more, since we just showed that the cumulative error is no worse than $2\varepsilon N$, its cumulative recorded count will be no worse than $f - 2\varepsilon N$ which is larger than εN, hence RoPE will report this value after composition. Finally, the space used at each collector is $\frac{1}{\varepsilon}\log(\varepsilon N_i)$ meeting our requirements.
Implementation: RoPE uses $\varepsilon = .01$. Micro-benchmarks show that frequency estimates are never worse off by more than εN and the space used is small multiples of $\log(\frac{N}{\varepsilon})$.

Hash Sketches to count Distinct Values.
Counting the number of distinct items in sub-linear state and in a single pass has canonically been a hard problem. Using only $O(\log N)$ space, hash sketches [9] computes this number approximately. That is, the estimate is a random variable whose mean is the same as the number of distinct values and the standard deviation is small.

The key technique involves uniformly random hashing the values. The first few bits of the hash value are used to choose a bit vector. From the remaining bits, the first non-zero bit is identified and the corresponding bit is set in the chosen bit vector. Such a hash sketch estimates the number of distinct values because $\frac{1}{2}$ of all hash-values will be odd and have their first non-zero bit at position 1, $\frac{1}{2^2}$ will do so at position 2 and so on. Hence, the maximum bit set in a bit-vector is proportional to the logarithm of the number of distinct values. Using a few bit-vectors rather than one guards against discretization error. The actual estimator is a bit more complex to correct for additive bias. For more details, please refer [9].

RoPE uses a distributed form of hash sketches. Each stat collector maintains local hash sketches and relays these bit vectors to the job manager. The job manager maintains global bit vectors, such that the i^{th} global bit vector is an *OR* of all the individual i^{th} bit vectors. By doing so, the global hash sketch is exactly the same as would be computed by one instrumentation point that looks at all data. Hence, RoPE retains all the properties of hash sketches.
Implementation: If the hash values are h bits long, where $h = O(\log N)$, and the first m bits choose the bit-vector, then there are 2^m bit vectors and the size of the hash sketch is $h * 2^m$ bits. RoPE uses $m = 6$ and $h = 64$. Hence, each task's hash sketch is 512B long. Our micro-benchmarks show that hash sketches retain precision even as the number of distinct values grows to 2^{40}.

3.1.2 Operation Costs

We collect the processing time and memory used by each task. A task is a multi-threaded process that reads one or more inputs from the disk (locally or over the network) and writes its output to the local disk. However, popular data-parallel frameworks can combine more than one operation into the same task (e.g. data extraction, decompression, processing). Since such operations run within the same thread and frequently exchange data via shared memory, the per-operator processing and memory costs are not directly available. But, per-operator costs are necessary to reason about alternate plans, e.g., reordering operators. Program analysis of the underlying code could reason about how the operators interact in a task, but this analysis can be hard because the interactions are complex, e.g., pipelining. Also, profiling individual operators does not scale to the large number of UDOs. Instead, RoPE uses a simple approach that only estimates the costs of the more costly operations.

The approach works as follows. First, tasks containing costly operations are likely to be costly as well. We pick stages with costs in the top tenth percentile as *expensive*. We only use the costs of the median task in each stage to filter the impact from failures and other runtime effects. Second, not all the operators in expensive tasks are expensive. So, for each operator, we compute a *confidence* score as the fraction of the stages containing the operator that have been picked as expensive. An operator will have a high confidence score only if it exclusively occurs in expensive tasks. Third, we compute the *support* of an operator as the number of distinct expensive stages that it occurs in. Finally, we estimate the cost of operators, that have high confidence and high support scores, as the average cost of the stages containing that operator.

To validate this approach, we profiled over 200K randomly chosen stages from the production cluster. It is hard to obtain ground truth at this scale. Hence, we corroborate our results on *succinctness* – only a few among all the operations should be expensive, and *coverage* – most of the expensive tasks should contain at least one expensive operation. This method identified 22.3% and 15.6% of the operators as expensive for CPU and memory costs respectively. This number was higher than expected because there are only a few generic operations but many UDOs. Most of the costly operations were UDOs. Further, 87.9% (92.4%) of the tasks picked as expensive based on their memory (cpu) cost contained at least one expensive operator. Hence, their cost can be explained by these operations. Very few contained more than one expensive operation. The succinct set of operations identified as expensive and the high coverage of this set makes us optimistic about this simple method. Fig 11 plots the relative cost values attributed to the expensive operations.

3.1.3 *Leading Statistics*

The ability to predict behavior of future operators is invaluable, especially for on-the-fly optimizations. Doing so precisely is hard. Rather than aspiring for precise predictions in all cases, RoPE collects simple leading statistics to help with typical pain points. We collect histograms and random samples on interesting columns (i.e. columns which are involved in where/group by clauses or joins) at the earliest collection points preceding the operator at which those columns will be used.

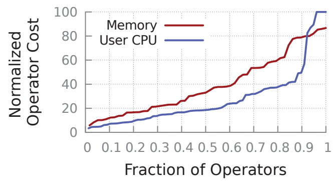


Figure 11: Normalized memory and user CPU costs for the operators identified as expensive.

We looked at several histogram generators, including equi-width, equi-depth and serial but ended up with a simpler, albeit less useful alternative. This is because none of the others satisfied our single pass and bounded memory criterion with a reasonable accuracy. For each interesting column, in an operator, we build a hash-based histogram with B buckets, that is hashed on a given column value (hash[column value] % $B \rightarrow$ bucket) and counts the frequency of all entries in each bucket. RoPE uses $B = 256$.

We also use reservoir sampling to pick a constant sized, random sample of the data flowing through the operator. For each interesting column, RoPE collects up to 100 samples but no more than 10KB.

3.2 Matching context to query expressions

As metadata to enable matching, with each stat collector we associate a hash-value that captures the location of the collector in the query graph. In particular, location in the query graph refers to a signature of the input(s) along with a topologically sorted chain of all the operators that preceded the stat collector on the execution plan. We colloquially refer to this as the query subgraph. RoPE uses 64 bit hashes to encode these query subgraphs.

3.3 Adapting query plans

We build on top of SCOPE Cloud Query Optimizer (CQO) which is a cost-based optimizer. CQO translates the user-submitted script into an expression tree, generates variants for each expression or group of expressions and finally chooses the query tree with the lowest cost. However, lacking direct knowledge of query context, the CQO uses simple apriori constants to determine the costs and selectivities of various operators as well as the data properties. By providing exactly this context, RoPE helps the CQO make better decisions.

RoPE imports statistics uses a stat-view matching technique similar to the analogous method in databases [4, 10, 18]. Statistics are matched in during the exploration stage in optimization, i.e. before implementing physical alternatives but after equivalent rewrites have been explored. The optimizer then propagates the statistics offered at one location to equivalent semantic locations, e.g., cardinality of rows remains the same after a sort operation and can be propagated. For expressions that have no direct evidence available, the optimizer makes-do by propagating statis-

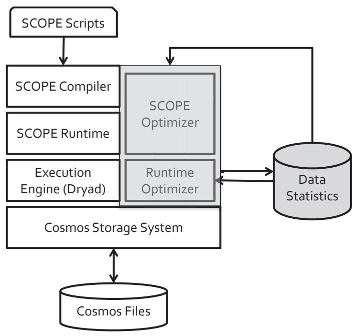


Figure 12: The RoPE prototype consists of 3 key components: a distributed statistics collection module, a pre-processor to the existing SCOPE compiler that provides matching functionality and a basic on-the-fly runtime optimizer.

tics from nearby locations with apriori estimates on the behavior of operators that lie along the path. Such uncertain estimates are deemed to be of lower quality and are used only when other estimates are unavailable.

We extended the optimizer to make use of these statistics. Cardinality, i.e., the number of rows observed at each collection point, helps estimate operator selectivity and compare reordering alternatives. Along with selectivity, the computation costs of operations are used to determine whether an operation is worth doing now or later when there is less work for it. Costs also help determine the degree of parallelism, the number of partitions, and which operations to fit within a task. Besides the choice of broadcast join, statistics also help decide when self-join or index-join are appropriate. Most of these optimizations are specific to the context of parallel executions. We believe that there is more to do with statistics than what RoPE currently does, but our prototype (§4) suffices to provide substantial gains in production (§5).

4. PROTOTYPE

The prototype collects all the statistics described in §3. Data stats are written to a distributed file system and the matching functionality (§3.2) runs as a pre-processing step of the job compiler. The statistics collection code is a few thousand lines of C#, which during code generation for each operator, gets placed into the appropriate location in the operator's dataflow. Note that not all operators collect statistics, and even when they do, they do not collect all types of statistics. Collected statistics are passed to the C++ task wrapper via reflection which piggybacks them along with task status reports to the job manager. Composing statistics required a few hundred lines of code in the job manager.

We allow the compiler to specify varying requirements across the columns. The constants also can change, to trade-off costs for improved precision, based on previous statistics or other information that the compiler has such as required accuracy of an estimate.

Parts of the code are production quality to the extent that all of our results here are from experiments that run in Bing's production clusters. Rather than implementing every optimization possible with the statistics that RoPE collects, we built a subset up to production quality code in order to deploy, run and gain experiences from Bing's production clusters. We acknowledge that our prototype is just that, and there is more benefit to be achieved.

Using these statistics required extensive changes in the SCOPE query optimizer involving several hundred lines of code spread over several tens of files.

5. EVALUATION

We built and deployed RoPE on a large production cluster that supports the Bing search engine at Microsoft.

5.1 Methodology

Cluster: This cluster constitutes of tens of thousands of 64-bit, multi-core, commodity servers; processes petabytes of data each day and is shared across many business groups. The core of the network is moderately oversubscribed and hence shuffling data across racks remains more expensive than transfers within a rack.

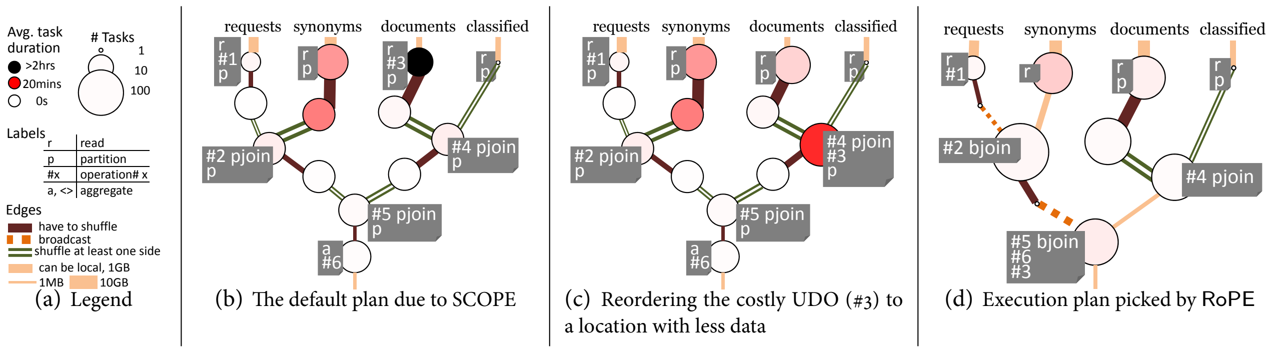
Workload Evaluated: We present results from evaluating RoPE on all the recurring jobs of a major business group at Bing and randomly chosen jobs from ten other business groups. The dataset has over 80 jobs. We repeat each job over ten times for each variant that we compare. The only modification we do to the jobs is to pipe the output to another location in the distributed file system so as to not pollute the production data. Though small, since we pick a large set of jobs, our dataset spans a wide range of characteristics. Latency of the unmodified runs varied from minutes to several hours. Tasks ranged from small tens to hundreds of thousands. The largest job had several tens of stages. User-defined operations are prevalent in the dataset, as is common across our cluster.

Compared Variants: For each job, we compare the execution plan generated by RoPE with the execution plan generated without RoPE. The latter is a strong strawman, since it uses apriori fixed estimates of operator costs and selectivities and is the current default in our production clusters. Early results from a variant that was equivalent to Hive over Hadoop, i.e., one that translated users' jobs into literal map-reduce execution plans, were significantly worse than our cluster's baseline implementation. Here, we omit those results.

Metrics: Our evaluation metrics are the reduction in job completion time and resource usage. Resources include cluster slot occupancy computed in machine-hour units, as well as the total amount of data read, written and moved across low bandwidth (inter-rack) network links.

5.2 A Case Study

Many plan changes can be performed given the statistics that RoPE collects. We report a case study that il-



(a) Legend (b) The default plan due to SCOPE (c) Reordering the costly UDO (#3) to a location with less data (d) Execution plan picked by RoPE

Figure 13: Case Study: Evolution of the execution plan as RoPE provides more statistics. See Table 2 for how well these plans do upon execution.

lustrates some of the changes to execution plans that achieved significant gains in practice. Aggregate results from applying RoPE to a wide variety of jobs are in §5.3.

Consider a job that processes four datasets. Let these be requests (R), synonyms (S), documents (D) and classified URLs (C). The goal is to compute how many requests (of a certain type) access documents (of a certain other type). Doing so, involves the following five operations:

1. Filter requests (R) by location. This operation has a selectivity of $\frac{1}{2000}x$.
2. Join requests (R) with synonyms (S). There are many synonyms per word, so the selectivity is $50x$.
3. Apply a user-defined operation (UDO) to documents (D). The selectivity is $\frac{1}{8}x$ but has a very large CPU cost per document.
4. Join documents (D) with the list of classified URLs (C). This has a selectivity of $\frac{1}{10}x$.
5. Join synonymized requests with documents containing classified URLs; has a selectivity of $\frac{4}{5}x$.
6. Finally, count number of requests per document URL.

The dataset sizes are R: 3GB, S: 18GB, D: 12GB, C: 160MB.

Figure 13(b) shows the plan computed by the unmodified optimizer which uses apriori estimates on data properties and costs. Each circle represents a stage, a collection of tasks that execute one or more operations which are listed in the adjacent label (see legend in 13(a)). Unlabeled stages are aggregates [14]. The size of the circle denotes the number of tasks allocated to that stage, in logarithmic scale. The color of the circle in linear *red*scale denotes the average time per task in that stage, darker implies longer. Figure 13(d)(13(c)) show the plan generated using (a subset of) the statistics provided by RoPE. We explain the figures as we go along. Table 2 shows how well these plans do when executed, the metrics are averaged over five different runs.

Recall that the compiler sets costs proportional to the data processed and sets selectivity based on the type of the operation– for instance, filters are assumed to be more selective than joins. These apriori estimates have mixed results. They pick the right choice for the operation on requests; the more selective filter operation (#1) is done before joining with synonyms (#2) (see Fig. 13(b) top left). However, for documents, the plan performs the user-defined filter (UDO) (#3) before the more selective join (#4) leading to a lot of wasted work. As we see in Table 2, due to the high cost of the UDO, this plan takes over $17x$ the time of the next best alternative.

By providing an estimate of the UDO's selectivity, RoPE lets the compiler join the documents dataset with the classified dataset first. Figure 13(c) depicts such a plan (see change in middle right). From Table 2, we see that since the UDO is applied on fewer data, the median execution time improves substantially. And, not many more tasks are needed since fewer documents through the UDO means fewer net work, and so the cluster hours decrease as well. But, by performing the UDO later, more data is shuffled across the network, since the join with classifieds is done on unfiltered documents.

When comparing these two plans, note that the thickness of the edges represent data volume moving between stages in logarithmic scale (see legend). Also, the color indicates the type of data movement. Dark solid lines denote data flow from a partition stage to an aggregate stage (many to many) which has to move over the network. Light solid lines indicate one-to-one data flow. Here, data-local placement can avoid movement across the network. Dotted lines indicate broadcast, i.e., the source stage's output is read by every task in the destination stage. Double edges indicate two source tasks per destination task, i.e., at least one of the sides has to be moved over the network.

With the UDO at a better place, the bottleneck moves to two new places. The average task duration of the stage with the UDO is very high (deep red). If the compiler knew the cost of the UDO, it could apportion more tasks to this stage to resolve this bottleneck. Similarly, even though requests (R) becomes small after applying the very selective filter (#1), the compiler is unaware and picks a pair-wise join for #2. This causes the large dataset S to be

Alternate Execution Plans	Performance				
	Latency (s)	Cluster Occupancy (s)	Cross Rack Shuffle (GB)	Reads+Writes To Disk (GB)	Tasks
Un-modified (Fig.13(b))	1x	1y	21.23	91.96	234
Push UDO to appropriate location (Fig. 13(c))	.057x	.201y	30.35	125.68	236
+ Replace pair-wise join with broadcast (not shown)	.016x	.227y	13.29	94.05	211
+ Balance (not shown)	.008x	.215y	12.82	90.75	341
RoPE (+push UDO even lower, little data,Fig. 13(d))	.006x	.139y	15.98	84.04	366

Table 2: Summarizes salient features of executing a typical job with and without RoPE. Overall, with RoPE latency reduces by almost 160X, while using 7X fewer cluster hours.

partitioned and shuffled across the network needlessly.

Figure 13(d) shows the plan where RoPE provides the compiler with the selectivity and costs of all the operations. Intermediate plans have been omitted for brevity. Table 2 shows that the median execution of this plan improves by another $9.5x$. A few changes are worth noting.

First, operation #2 is now implemented as a broadcast join. This not only saves cross rack shuffle and reads/writes to disk but also avoids partitioning both these datasets. Since pair-wise joins shuffle data across the network, such stages are more at risk from congestion induced outliers. We observed this with the default plan.

Second, given the high cost of the UDO (#3), the compiler instead of adding more tasks to the stage with operation #4 defers the UDO till even later, i.e., till after operations #5 and #6. Both these operations are net data reductive, however #6 is particularly so since it produces one row for each document url that is in the eventual output. The increase in cost from performing other operations on more data was lower than the benefits from performing the UDO on fewer data.

More optimizations are enabled recursively. The compiler realizes that both sides of the input for the join in operation #5 are small due to the cumulative impact of earlier operations, leading to the third improvement– implement operation #5 also as a broadcast join.

This leads to a fourth improvement that is subtle. Notice that operation #6, a *reduce* operation that needs data to be partitioned by document url to compute the number of times each URL occurs, now lies between operations #5 and #3 thereby eliminating one complete map-reduce!

To understand why, note that the join in operation #5 matches words in synonymized requests with words in the classified documents. Typically this join would require both inputs to be partitioned on words. However, here there is so little data that the left side (requests) can be broadcast and the right side (documents) can be partitioned in any way. Hence the right side is partitioned on url immediately after it is read to facilitate operation #6 and never re-partitioned thereby providing for coalescing many more operations into the same stage. The enabler for this optimization is *little data*– the later parts of most data parallel jobs can benefit from serial plans. Achieving the change, however, requires reasoning about the entire execution plan and not one stage at a time which is possible in RoPE due to the interface with a query optimizer.

Fifth, to offset the cost of the UDO, the compiler assigns more tasks (larger size) to the stage that now implements the UDO (along with operations #5 and #6). Doing so, is again a global change because the earlier operations have to partition the data enough ways. To benefit from not re-partitioning, the compiler implements the join in #4 with the same (larger) amount of parallelism.

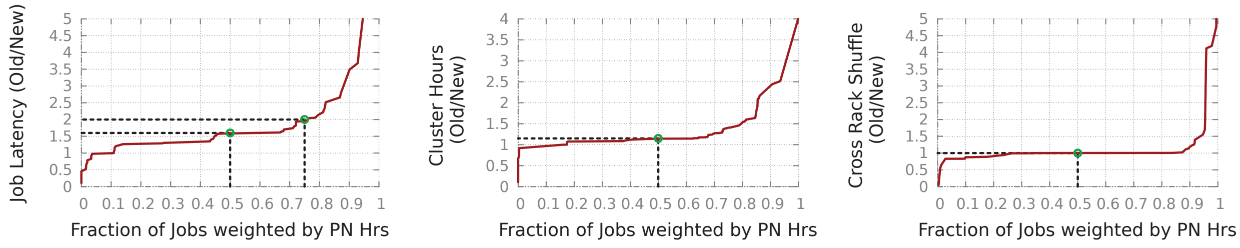
Finally, a potential change that was not performed is worth discussing. Since the classifieds dataset is small (C), it is possible to implement operation #4 also as a broadcast join. But the compiler chooses to not do that. The reason is that the documents have to be partitioned by url to facilitate the reduce in op #6. Either they are partitioned before the join op #5 or after. If the partition happens before, as is the case with the chosen plan, the large amount of work in the stage doing ops #5, #6 and #3 needs more parallelism resulting in more partitions, i.e., more tasks in op #4. But, the network costs of broadcasting C grow linearly with the number of tasks in op #4 offsetting the savings which is one additional read/write of C. Partitioning after op #5 would mean one more map-reduce on the critical path and just before the end of the job. Any outliers here would directly impact the job unlike outliers on the shuffle before op #4 which is not on the critical path since more work lies on the left side of the DAG.

Overall, we see that significant reductions result from optimizations building on top of each other. Unfortunately, the space of optimizations is not monotonic; some times using more tasks is better, whereas at other times pushing a filter after some operators but before some others is the best choice. By providing accurate estimates of code and data properties, RoPE is a crucial first step towards picking appropriate execution plans.

5.3 Aggregate results

Here, we present aggregated results across jobs in the dataset. These are runs of production jobs in a production cluster, so the noise from other jobs and other traffic on the cluster is realistic.

Figure 14(a) plots a CDF of the ratio between the job runtimes without and with RoPE. So, $y = 1$ implies no benefit. Samples below that line are regressions while those above indicate improvement. To compute a single metric from a disparate set of jobs, we weight job by the



(a) Reduction in Job Latency (b) Reduction in Cluster Hours Used (c) Reduction in data shuffled across racks

Figure 14: Aggregated results from production dataset (see 5.3)

total cluster hours that it occupies. All the results in this section share this format.

Figure 14(a) shows that 25% (50%) of the jobs speed up by over $2x$ ($1.5x$). Several significant jobs see over $5x$ reduction in their latencies. Upon inspection we find that all the samples under the $y = 1$ line are from two jobs. Both jobs have a small number of stages (and tasks). RoPE does not change their execution plans. Noise due to the cluster is the likely cause for lower performance. The vast majority of jobs see performance improvements. The reason is due to one or more of the optimizations described in the case study above.

Figure 14(b) shows that the latency savings due to RoPE are not from simply using more resources. In fact by avoiding wasteful work, RoPE speeds up jobs while reducing cluster usage. At the 50th (75th) percentile, jobs use $1.2x$ ($1.5x$) fewer cluster hours with RoPE.

Figure 14(c) shows that while the volume of cross rack shuffle is lower for some jobs, it stays the same for many and increases for only a few. This is expected since while some of the optimizations enabled by RoPE lower cross rack shuffle, others can increase it. Our optimization goal is to improve job latency and hence, on all the other metrics, we only indirectly constrain RoPE. Yet, we find that RoPE mostly achieves its gains by shuffling fewer amounts of data across the network. Figure 15(b) shows a similar pattern for the data read and written to disk. Figures 15(a) and 15(c) show that RoPE's plans mostly use fewer tasks and stages, though sometimes, for e.g., when offsetting the cost of UDOs, RoPE can use more tasks.

In summary, RoPE judiciously uses resources to improve job latency. Some gains accrue from avoiding wasted work, others from trading a little more of one type of resource for large savings on another while still others accrue from balancing the parallel plans.

6. RELATED WORK

Recent work on data-parallel cluster computing framework has mainly focused on solving issues that arise during the execution of jobs, by sharing the cluster [13, 23], tackling outliers [1], fairness vs. locality [25] and network scheduling [8]. Others incorporate new functionality such as the support for iterative and recursive control flow [20]. Orthogonally, RoPE generates better execution plans by leveraging data and execution insights.

The AutoAdmin project examined adapting physical database design, e.g., choosing which indices to build and which views to materialize, based on the data and queries.

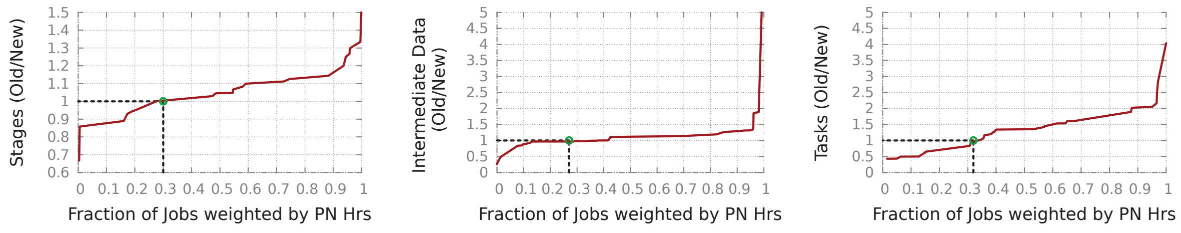
Closer to us, is the work that adapts query plans based on data. Kabra and DeWitt [16] were one of the earliest to propose a scheme that collects statistics, re-runs the query optimizer concurrently with the query, and migrates from the current query plan to the improved one, if doing so is predicted to improve performance. They mainly address the challenges of trading off re-optimization vs. doing actual work and of re-using the partial executions from the old plan to avoid wasting work that is already done. These challenges are easier in the context of map-reduce while collecting statistics is harder due to the distributed setting. Further, RoPE explores new opportunities arising due to the parallel nature of plans.

Eddies [2] adapts query executions at a much finer, per-tuple, granularity. To do so, Eddies (a) identifies points of symmetry in the plan at which re-ordering can happen without impacting the output, (b) creates tuple routing schemes that adapt to the varying selectivity and costs of operators. RoPE looks at a disjoint space of optimizations (choosing appropriate degrees of parallelism and operator implementations), which are not easily cast into Eddies' tuple routing algorithm.

Starfish [12] examines Hadoop jobs, one map followed by one reduce, and tunes low-level configuration variables in Hadoop such as io.sort.mb. To do so, it constructs a what-if engine based on a classifier trained on experiments over a wide range of parameter choices. Results show that the prescriptions from Starfish improve on developer's rules-of-thumb on non-traditional servers (e.g., fewer memory or cores). RoPE is complementary because (a) it applies to jobs that are more complex than a map followed by a reduce, (b) explores a larger space of plans and (c) uses a cost-based optimizer.

7. DISCUSSION

Recurring jobs are a first-order use case in our production system. We find that RoPE achieves meaningful improvements to such jobs. However, it is important



(a) Reduction in # of Stages (b) Reduction in intermediate writes to disk (c) Reduction in # of Tasks

Figure 15: Aggregated results from production dataset (see 5.3)

to note that statistics from a run only cover sub-graphs of operations used in that execution plan. This information may not suffice to find the optimal plan. After a few runs, we usually find all the necessary statistics since each run can explore new parts of the search space. However, plans chosen during this transition are not guaranteed to monotonically improve. In fact, there are no general ways to bound the worst case impact from plans chosen based on incomplete information. As a result, RoPE largely errs on the side of very conservative changes.

We defer to future work some advanced techniques that choose plans given uncertainity regimes over statistics or choose a set of plans, each of which has an associated validity range specified over statistics, and switch between these plans at runtime depending on the observed statistics [3]. Clearly these techniques are more complex than RoPE, the risks from picking worse plans are larger here, and to the best of our knowledge using these ideas in the context of parallel plans is an open problem.

8. FINAL REMARKS

Results from a deployment in Bing show that leveraging properties of the data, the code and their interaction significantly improves the execution of data parallel programs. The improvements derive from using statistics to generate better execution plans. Note that these improvements are mostly orthogonal to those from solving runtime issues during the execution of the plans (e.g., outliers, placing tasks). They are also in addition to those accrued by a context-blind query optimizer over literally executing the programs as specified by the users.

While RoPE leverages database ideas, we believe that the realization in the context of data parallel programs is interesting due to challenges that are new (e.g., distributed collection), or are more important in this context (e.g., user defined operations) and novel opportunities for improvement (e.g., recurring jobs, little data and the optimizations specific to parallel plans such as choosing degree of parallelism to achieve balance).

Acknowledgements

We thank Suman Nath, Peter Bodik, Andrew Ferguson, Jonathan Perry, our shepherd Roxana Geambasu and members of the UC Berkeley AMP Lab for comments that improved this work. The Cosmos team has built and keeps running the fabulous system without which this work would not have been possible.

References

[1] G. Ananthanarayanan, S. Kandula, A. Greenberg, et al. Reining in the Outliers in MapReduce Clusters Using Mantri. In *OSDI*, 2010.
[2] R. Avnur and J. M. Hellerstein. Eddies: continuously adaptive query processing. *SIGMOD Rec.*, 29, May 2000.
[3] S. Babu, P. Bizarro, and D. DeWitt. Proactive re-optimization. In *SIGMOD*, 2005.
[4] N. Bruno and S. Chaudhuri. Exploiting statistics on query expressions for optimization. In *SIGMOD*, 2002.
[5] R. Chaiken, B. Jenkins, P. Larson, et al. SCOPE: Easy and Efficient Parallel Processing of Massive Datasets. In *VLDB*, 2008.
[6] C. Chambers, A. Raniwala, F. Perry, et al. Flumejava: easy, efficient data-parallel pipelines. In *PLDI*, 2010.
[7] B. Chattopadhyay, L. Lin, W. Liu, et al. Tenzing a sql implementation on the mapreduce framework. In *VLDB*, 2011.
[8] M. Chowdhury, M. Zaharia, J. Ma, et al. Managing data transfers in computer clusters with orchestra. In *SIGCOMM*, 2011.
[9] M. Durand and P. Flajolet. Loglog counting of large cardinalities. In *ESA*, 2003.
[10] C. A. Galindo-Legaria, M. M. Joshi, F. Waas, and M.-C. Wu. Statistics on views. In *VLDB*, 2003.
[11] P. K. Gunda, L. Ravindranath, C. A. Thekkath, et al. Nectar: Automatic management of data and computation in datacenters. In *OSDI*, 2010.
[12] H. Herodotou, H. Lim, G. Luo, et al. Starfish: A self-tuning system for big data analytics. In *CIDR*, 2011.
[13] B. Hindman, A. Konwinski, M. Zaharia, et al. Mesos: a platform for fine-grained resource sharing in the data center. In *NSDI*, 2011.
[14] M. Isard et al. Dryad: Distributed Data-parallel Programs from Sequential Building Blocks. In *Eurosys*, 2007.
[15] M. Isard, V. Prabhakaran, J. Currey, et al. Quincy: Fair scheduling for distributed computing clusters. In *SOSP*, 2009.
[16] N. Kabra and D. J. DeWitt. Efficient mid-query re-optimization of sub-optimal query execution plans. In *SIGMOD*, 1998.
[17] Q. Ke, V. Prabhakaran, Y. Xie, et al. Optimizing data partitioning for data-parallel computing. In *HotOS*, 2011.
[18] P.-A. Larson et al. Cardinality estimation using sample views with quality assurance. In *SIGMOD*, 2007.
[19] G. S. Manku and R. Motwani. Approximate frequency counts over data streams. In *VLDB*, 2002.
[20] D. G. Murray and S. Hand. CIEL: a universal execution engine for distributed data-flow computing. In *NSDI*, 2011.
[21] C. Olston, B. Reed, U. Srivastava, et al. Pig Latin: A Language for Data Processing. In *SIGMOD*, 2008.
[22] A. Rasmussen, G. Porter, M. Conley, et al. Tritonsort: a balanced large-scale sorting system. In *NSDI*, 2011.
[23] A. Shieh, S. Kandula, A. Greenberg, et al. Sharing the data center network. In *NSDI*, 2011.
[24] A. Thusoo, J. S. Sarma, N. Jain, et al. Hive- a warehousing solution over a map-reduce framework. In *VLDB*, 2009.
[25] M. Zaharia, D. Borthakur, J. S. Sarma, et al. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In *EuroSys*, 2010.

Optimizing Data Shuffling in Data-Parallel Computation by Understanding User-Defined Functions

Jiaxing Zhang[†] Hucheng Zhou[†] Rishan Chen[†‡] Xuepeng Fan[†&] Zhenyu Guo[†]
Haoxiang Lin[†] Jack Y. Li [†§] Wei Lin[*] Jingren Zhou[*] Lidong Zhou[†]
[†]*Microsoft Research Asia* [*]*Microsoft Bing* [‡]*Peking University*
[&]*Huazhong University of Science and Technology*[*] [§]*Georgia Institute of Technology*

Abstract

Map/Reduce style data-parallel computation is characterized by the extensive use of user-defined functions for data processing and relies on data-shuffling stages to prepare data partitions for parallel computation. Instead of treating user-defined functions as "black boxes", we propose to analyze those functions to turn them into "gray boxes" that expose opportunities to optimize data shuffling. We identify useful functional properties for user-defined functions, and propose SUDO, an optimization framework that reasons about data-partition properties, functional properties, and data shuffling. We have assessed this optimization opportunity on over 10,000 data-parallel programs used in production SCOPE clusters, and designed a framework that is incorporated it into the production system. Experiments with real SCOPE programs on real production data have shown that this optimization can save up to 47% in terms of disk and network I/O for shuffling, and up to 48% in terms of cross-pod network traffic.

1 Introduction

Map/Reduce style data-parallel computation [15, 3, 23] is increasingly popular. A data-parallel computation job typically involves multiple parallel-computation phases that are defined by *user-defined functions* (or *UDF*s). The key to data-parallel computation is the ability to create *data partitions* with appropriate properties to facilitate independent parallel computation on separated machines in each phase. For example, before a reducer UDF can be applied in a reduce phase, data partitions must be *clustered* with respect to a reduce key so that all data entries with the same reduce key are mapped to and are contiguous in the same partition.

To achieve desirable data-partition properties, *data-shuffling* stages are often introduced to prepare data for parallel processing in future phases. A data-shuffling stage simply re-organizes and re-distributes data into appropriate data partitions. For example [45], before applying a reducer UDF, a data shuffling stage might need to perform a *local sort* on each partition, *re-partition* the data on each source machine for re-distribution to destination machines, and do a multi-way *merge* on re-distributed sorted data streams from source machines, all based on the reduce key. Data shuffling tends to incur expensive network and disk I/O because it involves all data [48, 26]. Our analysis of a one-month trace collected from one of our production systems running SCOPE [10] jobs shows that with tens of thousands of machines data shuffling accounts for 58.6% of the cross-pod traffic and amounts to over 200 petabytes in total. Data shuffling also accounts for 4.56% intra-pod traffic.

In this paper, we argue that reasoning about data-partition properties across phases opens up opportunities to reduce expensive data-shuffling. For example, if we know that data-partitions from previous computation phases already have desirable properties for the next phase, we are able to avoid unnecessary data-shuffling steps. The main obstacle to reasoning about data-partition properties across computation phases is the use of UDFs [24, 26]. When a UDF is considered a "black box", which is usually the case, we must assume conservatively that all data-partition properties are lost after applying the UDF. One of the key observations of this paper is that those "black boxes" can be turned into "gray boxes" by defining and analyzing a set of useful *functional properties* for UDFs in order to reason about data-partition properties across phases. For example, if a particular key is known to pass-through a UDF without any modification, the data-partition properties of that key will be preserved. Furthermore, we show how we can redefine the partitioning function in a data-shuffling phase to help preserve data-partition properties when UDFs are applied.

We make the following contributions in this paper. First, we define how a set of data-partition properties are related to data-shuffling in a simple and general UDF-centric data-parallel computation model. Second, we define how a set of functional properties for UDFs change the data-partition properties when the UDFs are applied. Third, we design a program analysis framework to identify functional properties for UDFs, and develop an optimization framework named SUDO to reason about data-partition properties, functional properties, and data shuffling. We further integrate SUDO into the production SCOPE optimization framework. Finally, we study the

[*]Services Computing Technology and System Lab, Cluster and Grid Computing Lab, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, 430074, China.

real workload on the SCOPE production system to assess the potentials for this optimization and provided careful evaluations of a set of example showcase applications. The study uses 10,000 SCOPE programs collected from production clusters; SUDO is able to optimize 17.5% of the 2,278 eligible programs which involve more than one data-shuffling stages. Experiments with real SCOPE programs on real production data have shown savings of up to 47% in terms of disk and network I/O for shuffling, and up to 48% in terms of cross-pod network traffic.

The rest of the paper is organized as follows. Section 2 introduces the system model that SUDO operates on, defines data-partition properties, and shows how they relate to data-shuffling. Section 3 defines functional properties of UDFs, describes how they affect data-partition properties when UDFs are applied, and presents the SUDO optimization framework that reasons about data-partition properties. Section 4 presents further optimization opportunities that expand on the framework in the previous section, by considering re-defining partition functions. Implementation details are the subject of Section 5, followed by our evaluations in the context of the SCOPE production system in Section 6. Section 7 discusses the related work, and we conclude in Section 8.

2 SYSTEM MODEL

A typical data-parallel *job* performs one or more transformations on large datasets, which consist of a list of *records*; each with a list of *columns*. A transformation uses a *key* that consists of one or more columns. For parallel computation, a dataset is divided into *data partitions* that can be operated on independently in parallel by separated machines. A data-parallel job involves multiple parallel-computation *phases* whose computations are defined by user-defined functions (UDFs). Following Map/Reduce/Merge [12], our model contains three types of UDFs: *mappers*, *reducers*, and *mergers*. By choosing low-level "assembly language"-like computation model, our techniques can be applied broadly: programs written in any of the high-level data-parallel languages, such as SCOPE [10], HIVE [40], PigLatin [33], and DryadLINQ [46], can be compiled into jobs in our model.

within-partition / cross-partition	none	contiguous	sorted
none	AdHoc	-	LSorted
partitioned	Disjoint	Clustered	PSorted
ranged	-	-	GSorted

Table 1: Data-partition properties defined in SUDO.

Certain *data-partition properties*, which are defined with respect to keys, are necessary before a computation phase can be applied. For example, a reducer or a merger requires that all the records with the same key are contiguous in the same partition. In general, data-partition properties specify behaviors within a partition and across partitions, as shown in Table 1. Within a partition, records are *contiguous* if all same-key records are stored together, or *sorted* if they are arranged by their keys. Across partitions, records are *partitioned* if same-key records are mapped to the same partition, or *ranged* if they are stored on partitions according to non-overlapping key ranges. Among the nine combinations (cells in Table 1), we focus specifically on the following six: AdHoc, LSorted, Disjoint, Clustered, PSorted, and GSorted. For example, GSorted means that the records are *sorted* within a partition and *ranged* across partitions. We do not include the rest because they are not important in practice based on our experiences; incorporating them into SUDO is straightforward.

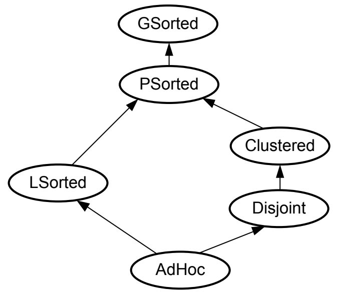


Figure 1: Data-partition property lattice in SUDO.

A data-partition property is *stronger* than another that it implies; for example, GSorted implies PSorted, which in turn implies Clustered. Such relationships are captured in the lattice shown in Figure 1, where a data-partition property is stronger than its lower data-partition properties. With this lattice, we can define *max* of a set of data-partition properties as the weakest one that implies all properties in that set. For example, *max* of Clustered and LSorted is PSorted. We define *min* analogously to max.

Data-shuffling stages are introduced to achieve appropriate data-partition properties by re-arranging data records without modifying them. A typical data shuffling stage consists of three steps: a *local-sort* step that sorts records in a partition with respect to a key, a *re-partition* step that re-distributes records to partitions via hash or range partitioning, and a multi-way *merge* step that clusters re-distributed records based on the key. Combinations of those steps are used to achieve certain properties; some requires taking all three steps, while others do not, depending on the data-partition properties before and after data shuffling. Figure 2 illustrates the relationship between data-partition properties and data-shuffling

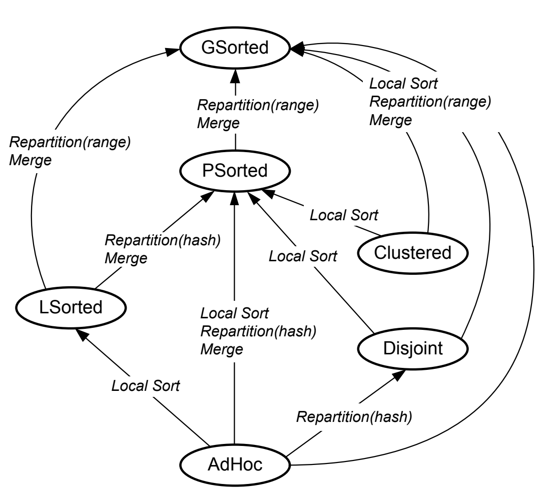


Figure 2: Transformation of data-partition properties using data shuffling.

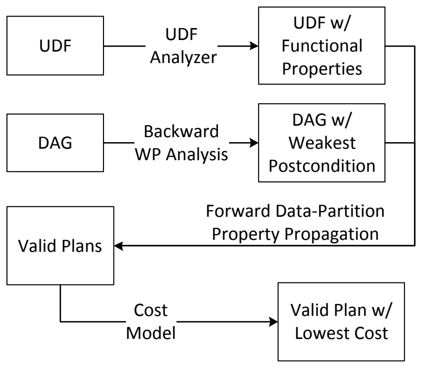

UDF
UDF Analyzer
UDF w/ Functional Properties
DAG
Backward WP Analysis
DAG w/ Weakest Postcondition
Valid Plans
Forward Data-Partition Property Propagation
Cost Model
Valid Plan w/ Lowest Cost

Figure 3: SUDO Optimization with Functional Properties.

steps. Note that because Clustered cannot be generated precisely through data-shuffling steps as the current implementation for the *merge* step uses merge-sort, SUDO always generates PSorted instead to satisfy Clustered.

In the rest of this paper, a data-parallel job is represented as a directed acyclic graph (DAG) with three types of vertices: *data vertices* that correspond to input/output data, each with an associated data-partition property; *compute vertices* that correspond to computation phasess, each with a type (mapper, reducer, or merger) and a UDF; and *shuffle vertices* that correspond to data-shuffling stages, each indicating the steps in that stage. The re-partitioning stages in shuffle vertices also specify whether hash or range partitioning is used. This DAG can be created manually or generated automatically by a compiler from a program in a high-level language, and allows us to flexibly define SUDO optimization scope. For example, we can use the same framework to analyze a pipeline of jobs or a segment of a job.

SUDO focuses on optimizing data shuffling by finding a *valid execution plan* with the lowest cost for a job *J*. The plan satisfies the following conditions: (i) the execution plan differs from *J* only at data-shuffling stages; (ii) for each computation phase, the input must have the necessary data-partition properties, i.e., data partitions must be Clustered for a reducer and PSorted for a merger; (iii) for a merger, all its input vertices must have the same data-partitioning (i.e., PSorted or GSorted on the same merge key); and finally, the execution plan preserves all data-partition properties of any output vertex.

Two SUDO optimizations are enabled by reasoning about how data-partition properties propagate through computation phases with UDFs. The first identifies unnecessary data-shuffling steps by using *functional properties* of UDFs to reason about data-partition properties, while the second further re-defines the partition function in a re-partitioning step of data shuffling to propagate certain data-partition properties for optimizations. We describe the first optimization in Section 3 and the second in Section 4.

3 Functional Properties

Data-shuffling stages in our model tend to be expensive as they often involve heavy disk and network I/O. They are added to satisfy data-partition properties for subsequent computation phases and to satisfy user requirements on output data. Although a preceding data-shuffling stage can result in certain data-partition properties, a computation phase with a UDF is not guaranteed to preserve those properties because traditionally, UDFs are considered "black boxes".

Our main observation for the first SUDO optimization is that, by defining appropriate functional properties, UDFs can be turned into "gray boxes" that expose how data-partition properties propagate across phases to facilitate the elimination of unnecessary data-shuffling steps. Figure 3 illustrates the overall flow of the SUDO optimization with functional properties. Given a job, we first extract and analyze all its UDFs to determine their functional properties. At the same time, we do a *backward WP analysis* to compute the *W*eakest *P*re-condition before each computation phase and the *W*eakest *P*ost-condition after each data-shuffling stage that maintains correctness as defined in Section 2. We then do a *forward data-partition property propagation* to generate valid execution plans with optimized data-shuffling stages, and then select the plan with the lowest cost according to a cost model. The rest of this section elaborates on this optimization flow.

3.1 Defining Functional Properties

A functional property describes how an output column that is computed by a UDF depends on the UDF's input columns. Because SUDO focuses on optimizing the data-shuffling stages that are added to satisfy data-partition properties, we are particularly interested in functional properties that preserve or transform data-partition properties. Ideally, those functional properties should be simple enough to be identified easily through automatic program analysis. Here we limit our attention to deterministic functions that compute a single output column from a single input column in one single record. A UDF might exhibit one functional property on one output column and another on another column; we focus on columns that are used as a reduce key, merge key, or re-partition key, as well as those used to compute those keys. A list of interesting functional properties for SUDO follows.

A *pass-through* function f is an identity function where the output column is the same as the corresponding input column. By definition, a reducer/merger is a pass-through function for the reduce/merge key. A pass-through function preserves all data-partition properties.

Function f is *strictly-monotonic* if and only if, for any inputs x_1 and x_2, $x_1 < x_2$ always implies $f(x_1) < f(x_2)$ (*strictly-increasing*) or always implies $f(x_1) > f(x_2)$ (*strictly-decreasing*). Examples of strictly-monotonic functions include normalizing a score (e.g., $\texttt{score}' = \lg(\texttt{score})/\alpha$), converting time formats (e.g., `DateTime.ToFileTime()`), adding common prefix or postfix to a string (e.g., supplementing "http://" and "/index.html" to the head and tail of a site), and any linear transformation (e.g., $y = a \cdot x + b$ where $a \neq 0$). A strictly monotonic function also preserves all data-partition properties, although the output column might be in a reverse sort-order.

Function f is *monotonic* if and only if, for any inputs x_1 and x_2, $x_1 < x_2$ implies $f(x_1) \leq f(x_2)$ (*increasing*) or $f(x_1) \geq f(x_2)$ (*decreasing*). Examples of monotonic functions include time-unit conversion (e.g., $\texttt{minute} = \lfloor \texttt{second}/60 \rfloor$) and substring from the beginning (e.g., "abcd" $\rightarrow$ "ab" and "ac123" $\rightarrow$ "ac"). Monotonic functions preserves sort-order within a partition, but are not guaranteed to preserve partitioned or ranged properties across partitions because two different input keys can be mapped to the same output key.

Function f is *one-to-one* if and only if, for any inputs x_1 and x_2, $x_1 \neq x_2$ implies $f(x_1) \neq f(x_2)$. Examples of one-to-one UDFs include reversing urls (e.g., "www.acm.org" $\rightarrow$ "org.acm.www") and MD5 calculation (assuming no conflicts). One-to-one functions do not preserve sort-order, but do preserve contiguity within a partition and the partitioned property across partitions. As a result, it preserves data-partition properties such as Disjoint and Clustered but downgrades GSorted and PSorted to Clustered.

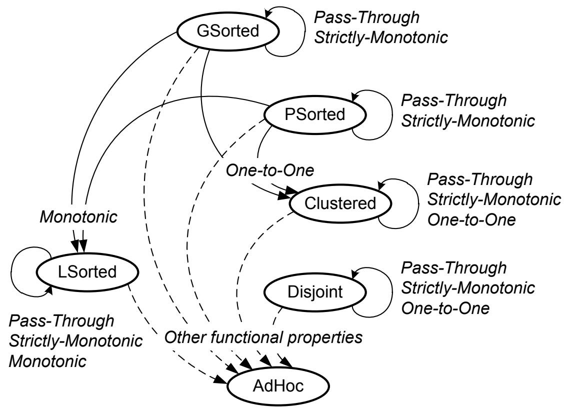


Figure 4: Data-partition property propagation through UDFs with various functional properties.

Figure 4 shows how data-partition properties propagate through UDFs with various functional properties, which are not chosen randomly. In fact, monotonic is sufficient and necessary for preserving LSorted; one-to-one is sufficient and necessary for preserving Clustered; and strictly-monotonic is sufficient and necessary for preserving GSorted. A proof is given in [47].

3.2 Identifying Functional Properties

SUDO allows users to annotate UDFs with appropriate functional properties. It further uses program-analysis techniques to infer properties automatically whenever possible, in a bottom-up approach inspired by *bddbddb* [1]. Because functional properties focus on the dependency relationship between an output column and its relevant input columns, SUDO applies program slicing [42] to extract a UDF's core function for inferring its functional property with respect to each output column of interest. The analysis then starts from *facts* about the input columns, and applies *deduction rules* to the low-level instructions as well as third-party library calls to infer functional properties recursively until a fixed point is reached. The process returns the final functional properties associated with the UDFs upon termination.

Figure 5 shows examples of facts and rules. A fact represents the property of a function between a variable in a UDF and an input column that the variable is computed from. For example, for a variable y and an input column t, such that $y = f(t)$ for some function f, $\texttt{One2One}(y,t)$ states that f is a one-to-one function. Figure 5 (line 1) defines the basic fact that every input col-

```
1  PassThrough(t,t)
2
3  _(y,t) :- ASSIGN y x, _(x,t)
4  StInc(z,t) :- ADD z x y, StInc(x,t), Inc(y,t)
5  StInc(z,t) :- ADD z x y, Inc(x,t), StInc(y,t)
6
7  One2One(y,t) :- MD5 y x, One2One(x,t)
8
9  StInc(x,t) :- PassThrough(x,t)
10 One2One(x,t),Inc(x,t) :- StInc(x,t)
11 One2One(x,t),Dec(x,t) :- StDec(x,t)
12 Func(x,t) :- _(x,t)
13 Inc(x,_),Dec(x,_) :- Constant(x)
```

Figure 5: Examples of facts and deduction rules in a datalog format. `StInc`, `StDec`, `Inc`, and `Dec` stand for strictly-increasing, strictly-decreasing, increasing, and decreasing, respectively.

umn is considered a `PassThrough` function over itself.

Deduction rules infer the functional property of an instruction's output operand from the functional properties of its input operands. SUDO introduces a set of deduction rules; examples are shown in Figure 5 (lines 3-5). The first rule (line 3) states that, for `ASSIGN` instructions, the functional property of the input operand x can simply be propagated to the output operand y (the _ symbol means any functional property). The second and third rules state that, for `ADD` instructions, the output operand is strictly-increasing as long as one of the input operands is strictly-increasing, while the other is increasing.

Besides instructions, UDFs also call into functions in third-party libraries. SUDO either applies the deduction rules directly to the instructions in the library calls, or treats these function calls simply as "instructions" and provide deduction rules manually. SUDO has an accumulated knowledge base with manually provided deduction rules for commonly used library calls. Those manual annotations are particularly useful for cases where the automatic inference runs into its limitation. For example, the functional property for `MD5` (line 7) cannot be inferred automatically.

We also have rules that encode the relations among functional properties, where one function property might imply another. Examples of such rules are shown in Figure 5 (lines 9-13). We use `Func` as the weakest functional property that satisfies no specific constraints and also introduce `Constant` as a pseudo functional property for constant values, which is both *increasing* (`Inc`) and *decreasing* (`Dec`).

3.3 Backward WP Analysis

Based on the definition of a valid execution plan, we must figure out the requirement on each data-shuffling stage for validity. This is done through the following backward WP analysis in a reverse topological order of a given job. When visiting a new vertex, the backward WP

```
procedure OnVisitVertex($v$)
  $v.WPrecondition \leftarrow AdHoc$
  if $v.Type = UDF$ then
    if $v.UDF = Reducer$ then
      $v.WPrecondition \leftarrow Clustered$
    else if $v.UDF = Merger$ then
      $v.WPrecondition \leftarrow PSorted$
    end if
  else if $v.IsOutputVertex$ then
    $v.WPrecondition \leftarrow v.GetOutputDataProperty()$
  else
    $v.WPostcondition \leftarrow$
      $v.OutVertices.Select(ov => ov.WPrecondition).Max()$
  end if
end procedure
```

Figure 6: Algorithm for backward WP analysis.

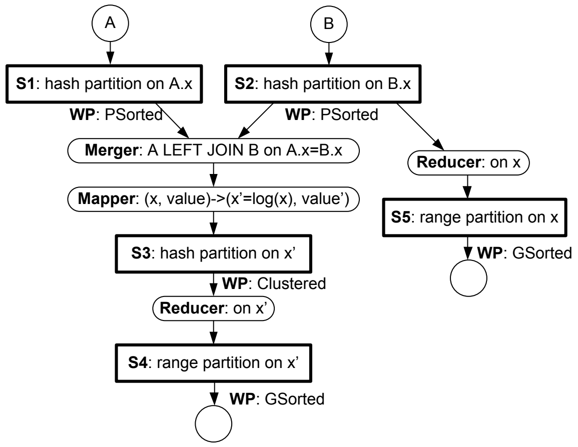

A
B
S1: hash partition on A.x
S2: hash partition on B.x
WP: PSorted
WP: PSorted
Merger: A LEFT JOIN B on A.x=B.x
Reducer: on x
Mapper: (x, value)->(x'=log(x), value')
S5: range partition on x
WP: GSorted
S3: hash partition on x'
WP: Clustered
Reducer: on x'
S4: range partition on x'
WP: GSorted

Figure 7: A sample data-parallel job.

analysis computes weakest precondition and postcondition as shown in Figure 6; the weakest postcondition associated with a data-shuffling stage is the data-partition property required on the result of that stage.

Figure 7 depicts a sample compiled DAG from a SCOPE job: circles correspond to data vertices; rectangles correspond to data-shuffling stages, and rounded rectangles correspond to computation phases with UDFs. SUDO firstly analyzes the UDFs in the job, and annotates them with functional properties. In this case, both the merger and the reducers are pass-through functions, and the mapper is a strictly-monotonic function (using *log*). Then it runs backward WP analysis to get the weakest postcondition for all data shuffling stages: this results in GSorted for S4 and S5, Clustered for S3, PSorted for S1, and *max*(PSorted, Clustered) = PSorted for S2.

3.4 Forward Property Propagation

Once SUDO completes the backward WP analysis, it tries to find all valid execution plans accord-

```
procedure ForwardExplorer ($currGraph, traverseSuffix$)
  while $traverseSuffix.IsNotEmpty()$ do
    $v \leftarrow traverseSuffix.RemoveFirst()$
    if $v.InVertices.Count = 0$ then
      $inputDP \leftarrow v.Input.GetDataProperty()$
    else if $v.InVertices.Count = 1$ then
      $inputDP \leftarrow v.InVertices[0].CurrentPostDP$
    else
      if $!v.ValidateMerge()$ then
        return
      end if
      $inputDP \leftarrow v.InVertices[0].CurrentPostDP$
    end if
    if $v.Type = UDF$ then
      $v.CurrentPostDP \leftarrow$
        $DPPropertyTransition(inputDP, funcProperty)$
    else
      for $prop$ in $\{p | p >= v.WPostcondition\}$ do
        $v.CurrentPostDP \leftarrow prop$
        $v.SSteps \leftarrow GetShufflingSteps(inputDP, prop)$
        $ForwardExplorer(currGraph, traverseSuffix)$
      end for
    end if
  end while
  $Plans.Add(currGraph)$
end procedure
```

Figure 8: Algorithm for enumerating all valid plans.

ingly through forward data-partition property propagation. This process tracks output data-partition property in a `CurrentPostDP` field for each vertex and discovers valid query plans along the way. The goal is to set the needed steps in each data-shuffling stage in order to generate valid execution plans. This is done through a recursive procedure `ForwardExplorer` (shown in Figure 8), which takes the current execution graph (`currGraph`) and the current suffix of the topologically ordered list of vertices (`traverseSuffix`). The procedure also uses three primitives: `DPPropertyTransition` takes an input data-partition property and a functional property, and outputs the resulting data-partition property based on the propagation graph in Figure 4; `GetShufflingSteps` takes an input data-partition property and an output data-partition property, and outputs the needed data-shuffling steps according to Figure 2; `ValidateMerge` checks whether input vertices of a merger phase all conform to the same data-partition property (PSorted or GSorted on the merge key).

Using the same example in Figure 7, we show how the algorithm can create a different execution plan: the `CurrentPostDP` is set to AdHoc for the input data vertices and PSorted for S1 and S2. Because the Merger is a pass-through function, its `CurrentPostDP` is set to PSorted. The `CurrentPostDP` is also set to PSorted after the Mapper because it is strictly-monotonic. Because PSorted implies Clustered, which is the weakest postcondition for S3, all steps of S3 can be removed to produce an alternative valid plan.

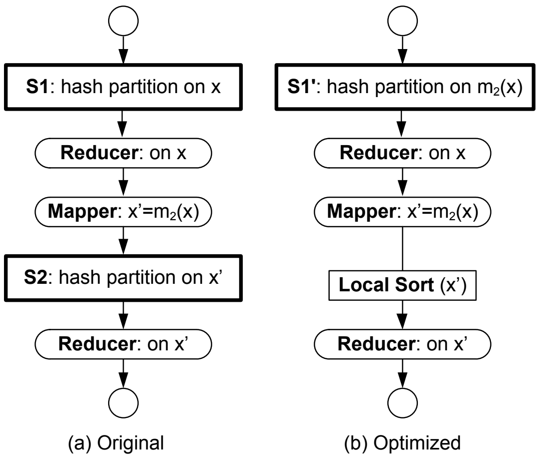


Figure 9: Chained shuffling optimization.

For simplicity, this algorithm enumerates all valid execution plans. It is conceivable to add heuristics to prune the candidates if there are too many valid execution plans to evaluate. All valid executions are evaluated based on the cost model, where the one with the lowest cost will be chosen.

4 Re-Defining Partitioning Key

Not all UDFs have the desirable functional properties for preserving data-partition properties, especially when we conservatively assume the input data can be arbitrary. SUDO can further leverage the ability to re-define a partitioning key to apply some constraint to the input data so as to preserve certain data-partition properties (e.g., Disjoint) for optimizing data-shuffling. These mechanisms further increase the coverage of the optimization framework described in Section 3. This section describes how we can re-define partitioning keys to preserve data-partition properties.

Considering the simple case in Figure 9 with two data-shuffling stages S_1 and S_2, where S_1 does a hash re-partitioning on key x, S_2 does a hash re-partitioning on key x', and mapper m_2 is not one-to-one, we cannot eliminate steps in S_2 using the algorithm discussed in Section 3 because m_2 does not have the needed functional property. It is however possible to re-define the partitioning key in S_1 by taking into account m_2 and S_2, in order to eliminate some steps in S_2. For example, if m_2 maps each x to a single x', SUDO can apply hashing on $m_2(x)$ for S_1, rather than on x directly. Then, this hash re-partitioning ensures not only that all records with the same x are mapped to the same partition, but also that all records with the same x' after applying m_2 are mapped to the same partition. This can help eliminate the re-partitioning step in S_2.

It is worth pointing out that there are side effects in this optimization, although it reduces the amount of total network I/O by eliminating a later re-partitioning step. The re-partitioning in S'_1 is slightly more expensive as it has to invoke m_2 on all input records; the number of these records might far exceed the number in the later mapper phase with m_2 because of the reducer on x. To reduce this extra overhead, SUDO applies program slicing to get a simpler function, as done in the analysis of functional properties. Also, the new partitioning might lead to data skew that does not exist in the original plan because of the different partitioning key used. Again, a cost model is used to assess whether or not to apply this optimization.

We can easily generalize this type of optimizations to a chain of data-shuffling stages. $S_1, S_2, \ldots, S_N$ is a chain of data shuffling with hash re-partitioning, where before each S_i (except S_1) there is a mapper with UDF m_i ($i = 2 \ldots N$). To guarantee that a single re-partitioning causes all the keys in later phases to be partitioned appropriately, we can construct partition function in S_1 as a hash on $(m_N \ldots m_3 m_2(x))$ where x is the initial input key.

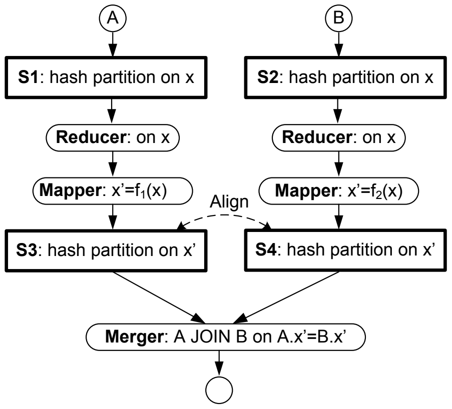


Figure 10: Joined shuffling optimization.

Two other data-shuffling patterns, *joined shuffling* and *forked shuffling* are more complicated. Joined-shuffling is widely used to implement a JOIN to correlate records in data-parallel programs. Figure 10 shows one such case. Shuffling stages S_3 and S_4 are required and inserted by the merger phase. This pattern can be considered as the merge of multiple chained data shuffling: in this example, one formed by S_1, f_1, and S_3, while the other by S_2, f_2, and S_4. Separately, SUDO can use a hash function on $f_1(x)$ for S_1 and a hash function on $f_2(x)$ for S_2 in order to remove re-partitioning in S_3 and S_4. Due to merger, the two chains must be aligned in that the same key will be mapped to the same partition. This is easily achievable as long as SUDO applies the same hashing to $f_1(x)$ and $f_2(x)$.

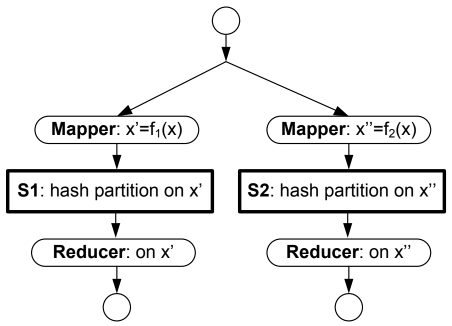


Figure 11: Forked shuffling optimization.

Figure 11 shows an example of forked shuffling, where an input is consumed by two separate threads of processing. In the figure, f_1 and f_2 are two mapper functions that map a record key from x to x' and x'', respectively. Two data-shuffling stages S_1 and S_2 perform hash re-partitioning on x' and x'', respectively. It is conceivable that SUDO can perform one data-shuffling by hashing on a function of x, to create disjoint data partitions both for $f_1(x)$ and for $f_2(x)$. That is, the data-shuffling must guarantee that, if $f_1(x_1) = f_2(x_2)$, then x_1 and x_2 are mapped to the same partition by S_0. For example, given $f_1(x) = \lfloor x/2 \rfloor$ and $f_2(x) = \lfloor x/3 \rfloor$, stage S_0 should use $\lfloor x/6 \rfloor$ as the re-partitioning key. Such a function might not always exist, as is the case with $f_1(x) = \lfloor x/2 \rfloor$ and $f_2(x) = \lfloor (x+1)/2 \rfloor$. Constructing the symbolic function automatically is challenging. We resort to recording known patterns and recognizing them through pattern matching on a sliced data-dependency graph. Currently, SUDO includes the following patterns: $f(x) = \lfloor x/a \rfloor$, $f(x) = x \ mod \ a$, where x is an integer.

SUDO creates new execution options with re-defined partitioning keys: they can be regarded as special mechanisms to preserve certain data-partition properties. Those options can be integrated into the framework described in Section 3 to create more valid execution plans, which will be evaluated using a cost model.

5 IMPLEMENTATION

We have implemented SUDO based on and integrated into the SCOPE compiler [10] and optimizer [48]. SCOPE is a SQL-like scripting language for data-parallel computation. The SCOPE optimizer uses a transformation-based optimizer to generate efficient execution plans. The optimizer leverages many existing work on relational query optimization and performs rich and non-trivial query rewritings that consider the input script in a holistic manner. The main contribution of the

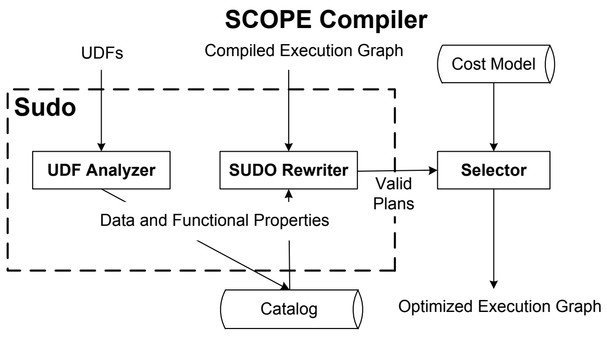


Figure 12: SUDO architecture.

SUDO implementation is to add the support of reasoning about functional properties, data-partition properties, and data shuffling into the current optimizer: without understanding UDFs, the system cannot derive any structural properties and thus potentially miss important optimization opportunities. With SUDO, functional properties between input and output columns are identified and integrated into the optimization framework. It enables efficient property derivation and allows the optimizer to optimize query plans with UDFs effectively.

Figure 12 depicts an architectural overview of SUDO, its components, and their interactions with the existing components in the SCOPE compiler and optimizer. The *UDF analyzer* and the *rewriter* are the two main SUDO modules. The UDF analyzer extracts the functional properties of the UDFs, while the *rewriter* generates optimized execution plans by leveraging the functional properties. The current implementation for the two modules contains 8,281 and 1,316 lines of C# code, respectively. The execution plans are further fed to the *selector*, which uses a cost model to find the best one among them.

5.1 UDF Analyzer

We have implemented the UDF analyzer at the high-level IR (HIR) of the Phoenix framework [35] shipped with Visual Studio 2010, together with the bddbddb engine [1]. Phoenix is a framework for building compiler-related tools for program analysis and optimizations. It allows external modules to be plugged in with full access to the internal Phoenix Intermediate Representation (IR). With Phoenix, the analyzer feeds the instructions and the library calls for a given UDF to the bddbddb engine, together with the deduction rules. The engine then applies the deduction process, as discussed in Section 3.2.

We describe in further detail the rules in SUDO. In the extracted IR excluding opcode `CALL`, we select top 8 unary operators and 7 binary operators in terms of frequency of use. These operators account for 99.99% of operator uses (again excluding `CALL`). We have a total of 28 rules for those operators. Coming up with those 28 rules requires some care. First, if done naively, we might have many more rules. We have reduced the number with the following two approaches: (i) some instruction type, such as `ASSIGN` and `BOX`, belong to the same *equivalent class* as they share the same set of deduction rules. (ii) binary operators (e.g., the `ADD` operator shown in Figure 5 (b)) usually requires more than one rule. We manually introduce several pseudo operators to convert some to others. For example, we add `NEGATE` and `RECIPROCAL` in order to convert `SUBSTRACT` and `DIVIDE` to `ADD` and `MULTIPLY` respectively, thereby reducing the total number of rules.

Second, *constraints* are often needed in the rules for precision. The constraints could be on some aspects of the operands, such as their types and value ranges. For example, the `CONVERT` operator is used to convert numbers between different types. Converting a number from a type with a smaller byte size to one with a larger size (e.g., from `int` to `double`) preserves its value; that conversion is considered a pass-through function. This is *not* the case for the opposite direction. SUDO extracts operand types and makes the rules type-sensitive with the type constraints embedded to handle these cases.

Finally, the UDFs contain loops and branches. The value of an operand may come from any one of its input operands defined in any of the branches. SUDO introduces the `INTERSECT` operator with the rule stating that the output operand has a certain property if both its input operands have the same functional property.

For the 157 system calls in the extracted IR of mscorlib and ScopeRuntime, we set 73 rules for 66 unary calls and 10 binary calls. The remaining system calls are marked `Func` (as in line 12 of Figure 5). For the 891 third-party calls (with no source code), we select top 10 most-frequently used and wrote 16 rules for them. Most of them are string related (e.g., `UrlReverse`, `PrefixAdd`, and `String.Concat`), while others are mostly math related.

5.2 SUDO Rewriter

SUDO rewriter generates all valid execution plans using the algorithms presented in Section 3, as well as the mechanisms described in Section 4. The implementation of the algorithm for enumerating all of the valid plans described in Section 3 is straightforward. In practice, the techniques in Section 4 plays an important role in uncovering optimization opportunities even after manual performance tuning and optimizations. The goal of SUDO rewriter is in principle similar to that of the SCOPE optimizer in that they are both enumerating all valid plans for cost estimation, although with key differences; for example, the SUDO rewriter works at the "physical" level, while the SCOPE optimizer starts with logical relational operators. For ease of implementation, the current SUDO rewriter simply takes as input the best physical execution plan from the SCOPE optimizer. The results from the SUDO rewriter are then assessed based on the internal cost model of the SCOPE optimizer. This simple integration might lead to sub-optimal results as two optimization phases are carried out separately. We are in the process of integrating SUDO into the SCOPE optimizer to reason about functional properties and structured data properties in a single uniform framework, and seamlessly generates and optimizes both serial and parallel query plans.

6 EVALUATION

In this section, we use real traces in a SCOPE production bed to assess the overall potential for SUDO optimizations and evaluate in details the effectiveness of those optimizations on representative jobs (pipelines).

6.1 Optimization Coverage Study

We have studied 10,099 jobs collected over a continuous period of time from a production bed with tens of thousands machines. The study aims at revealing the distribution of functional properties for the UDFs in those jobs and gauging the optimization opportunities for SUDO to leverage those properties.

Property	UDF funcs (#)	Ratio(%)
Pass-through	1,998,819	84.73
Strictly-increasing	147,820	6.27
Strictly-decreasing	0	0.00
Increasing	138	0.00
Decreasing	0	0.00
One-to-one	1,758	0.08
Func	210,544	8.92
Sum	2,359,079	100

Table 2: Statistics on functional properties.

We first look at the computation for each output column in each UDF to infer its functional property. Our analysis is limited to handling only computation where an output column in a row depends on a single input column in a single input row. Among the 236,457 UDFs and 3,000,393 output columns, 2,359,079 (78.63%) output columns satisfy this constraint. We then carry out our study on the 2,359,079 functions that produce those output columns.

For each functional property, Table 2 reports the number of functions that have this functional property and the percentage. For a function that satisfies multiple functional properties, only its strongest functional property is counted. For example, for a strictly-increasing function, it is counted only as strictly-increasing, but not as increasing. Pass-though functions clearly dominate and accounts for 84.73%, followed by strictly-increasing functions with 6.27%. A small fraction (0.08%) of the functions are one-to-one. About 8.92% of the functions do not have any of the functional properties SUDO cares about. Surprisingly, we do not find any (strictly-) decreasing functions.

We further run our optimizer to see how many jobs SUDO can optimize by leveraging the identified functional properties. This study focuses only on jobs with more than one data-shuffling stages. This eligible job set contains a total of 2,278 (22.6%) jobs. Among all the eligible jobs, SUDO is able to optimize 17.5% of them.

It is worth noting that the original SCOPE compiler supports user annotations of the pass-through functional property. In practice, 6% of the eligible jobs have been annotated, and 4.6% are actually optimized. Our automatic analysis engine is shown to have discovered significantly more optimization opportunities.

6.2 Optimization Effectiveness Study

To study the optimization effectiveness on individual jobs (pipelines), we select three important web search related SCOPE jobs. All the jobs are running on the same cluster as the one we collected the trace from for our UDF study. The number of machines used in each job depends on the size of the input data. Table 3 summarizes the optimization details for each case, while Table 4 gives the detailed performance comparisons between the original versions and the optimized ones. In this section, we describe these cases with their optimizations and then discuss the performance numbers together for ease of comparison.

6.2.1 Anchor Data Pipeline

Hyperlinks in web pages form a web graph. Anchor texts associated with hyperlinks, together with the web graph, are valuable for evaluating the quality of web pages and other search-relevance related metrics. One of the basic anchor-data pre-processing jobs is to put the anchors that point to the same page together (using a data-shuffling stage), and de-duplicate the anchors with the same text. The job further outputs the reversed url and the anchor text pairs, e.g., ("org.acm.www/sigs", anchor text) instead of ("www.acm.org/sigs", anchor text), as this reversed url format is the de-facto representation of a url as urls of the same domain are laid out contiguously in that format to enable simple domain-level aggregations.

The optimization opportunity comes when other jobs consume the output of this job. Figure 13 shows an example where the second job *Job2* tries to count the words in the anchor text for each url. Before optimization, Job2 has to insert a data shuffling stage (*S*2) to group the

Case	OptStages	FuncProperty	PreservedDP	RPF Stages	PreservedDP+	EndShuffSteps
§ 6.2.1	S2	One-to-One	Clustered	{∅}	Clustered	{∅}
§ 6.2.2	S2,S3	Increasing,Increasing	LSorted,LSorted	{S1}	PSorted,PSorted	{∅},{∅}
§ 6.2.3	S3,S4, S5,S6	None,None, Pass-through,Pass-through	AdHoc,AdHoc, GSorted,GSorted	{S1,S2}	Disjoint,Disjoint, GSorted,GSorted	{LS},{LS}, {∅},{∅}

Table 3: Optimization detail for the three cases. The columns represent the section reference, the shuffling stages that are optimized (*OptStages*), the functional properties of the UDFs before the shuffling stages, preserved data-partition property by the UDFs (*FuncProperty*), shuffling stages having their partition functions re-defined (*RPF Stages*), data-partition properties preserved with both *FuncProperty* and *RPF*, and the final steps for each in *OptStages*. *LS* stands for local sort.

data by url. However, this operation is redundant because url reversing is a one-to-one function, which preserves the Clustered property. SUDO recognizes this functional property and eliminates stage *S*2.

6.2.2 Trend Analysis Pipeline

Trend analysis is a way to understand how things change over time, which is important for many search related applications as well as being a stand-alone service. One kind of trend-analysis job collects the $\langle term, time, aspect\rangle$ tuples in the search-query logs, where *term* is the search keyword, *time* is when the query is submitted, and *aspect* is one of the search query's property (e.g., its market), and aggregates various aspects' occurrences at different time-scales such as hours, days, and months. For instance, we can know the top three market along the years for a specific brand using trend analysis.

Figure 14 shows a pipeline of trend-analysis jobs. Job1 pre-processes the input query logs and aggregates the entries within the same second for all aspects we need. Based on this result, Job2 calculates the $\langle term, time, market\rangle$ distribution over days, and Job3 calculates the $\langle term, time, intent\rangle$ distribution over weeks, where intent is obtained from user-click-behavior of the queries and tells whether the search query is for information, business, or other purposes. Before optimization, each job requires a data-shuffling stage. SUDO merges the three shuffling stages into one and redefines the partition key in Job1. The function ensures that the seconds within the the same week are always on the same partition, which guarantees the Disjoint property even after the two mapper functions that converts seconds to days and weeks, respectively. Besides, because the time conversion function is increasing, LSorted is preserved

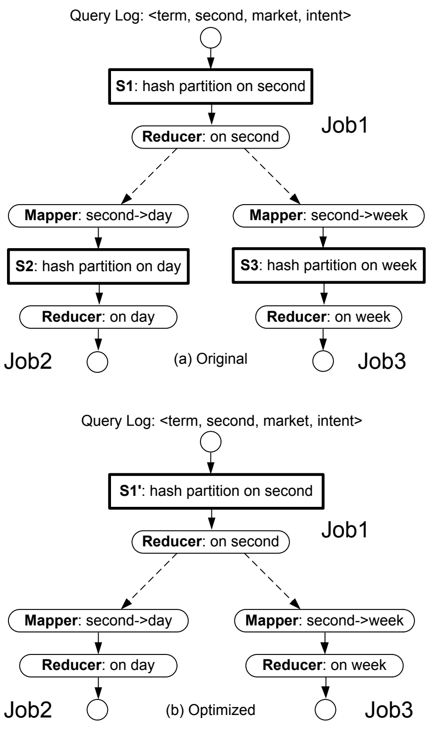


Figure 14: Optimization on the trend-analysis pipeline. The two data-shuffling stages on new time-scales are merged into the previous shuffling.

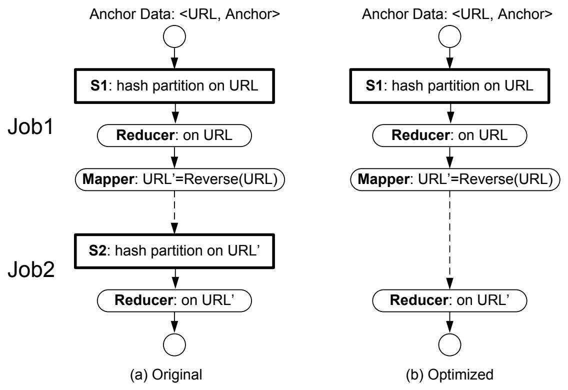

Anchor Data: <URL, Anchor>
S1: hash partition on URL
Job1
Reducer: on URL
Mapper: URL'=Reverse(URL)
S2: hash partition on URL'
Job2
Reducer: on URL'
(a) Original
Anchor Data: <URL, Anchor>
S1: hash partition on URL
Reducer: on URL
Mapper: URL'=Reverse(URL)
Reducer: on URL'
(b) Optimized

Figure 13: Optimization on the anchor data pipeline. The redundant shuffling stage in Job2 is eliminated.

and the local-sort operations in Job2 and Job3 are eliminated. Together, the optimization eliminates two shuffling stages, and ensures PSorted property before each reducer function in the three jobs.

6.2.3 *Query-Anchor Relevance*

Search queries and anchors are two term sets that can be bridged by urls. Looking at the joined data set is important to improve the search quality. If a query happends to result in a url that an anchor points to, then the query and the anchor are very likely to be relevant. For instance, if the word "China" appears frequently in query with result URL `example.org/a.html`, and the word "Emerging Market" appears in the anchor that points the same `example.org/a.html`, then "China" and "Emerging Market" are relevant. Furthermore, if these two words appear in `example.org` many times and their pointing-to urls also overlap a lot, they have a high relevance.

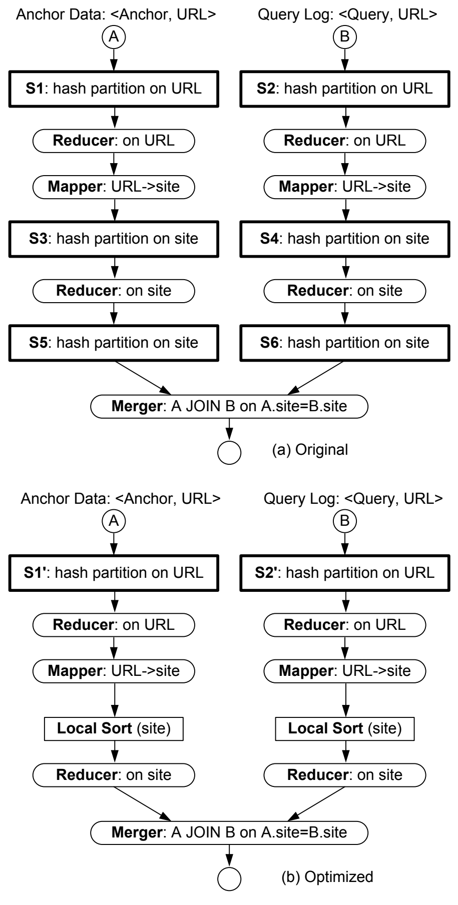

Anchor Data: <Anchor, URL>
Query Log: <Query, URL>
A
B
S1: hash partition on URL
S2: hash partition on URL
Reducer: on URL
Mapper: URL->site
S3: hash partition on site
S4: hash partition on site
Reducer: on site
S5: hash partition on site
S6: hash partition on site
Merger: A JOIN B on A.site=B.site
(a) Original
S1': hash partition on URL
S2': hash partition on URL
Local Sort (site)
(b) Optimized

Figure 15: Optimization on anchor-query relevance analysis jobs. Four shuffling stages are saved.

Figure 15 shows a job that aggregates anchor data as well as query logs, correlates them via sites, and applies some learning models for relevance study (omitted in the figure). The job first groups anchor texts on url (around $S1$), and reduces that into $\langle$url, map$\langle$*term*,*frequency*$\rangle\rangle$, where *term* is a keyword inside anchor texts and *frequency* is the number of times that the term occurred in these anchors. Then it converts urls to sites using a mapper, and groups the records on the same sites ($S3$). Inside the reducer, it computes an aggregated frequency for each term based on the data for different urls (within the same site). The job applies almost the same algorithm to the query logs too. After that, it joins these two output datasets on site for further study with two further shuffling stages ($S5$ and $S6$). There are in total six shuffling stages in the job; SUDO applies its optimization and makes it two, as shown in the figure. The new shuffling stages partitions the input data according to the sites they belong to, so as to keep the Disjoint property along the data flow. However, because the desired data-partition property for reduce on site is not satisfied (the mapper function which converts url to site does not preserve any data-partition property), SUDO partially eliminates stage $S3$ and $S4$ with a local-sort operation only. Because the reducer on site does not change the site key, the last two data-shuffling stages can be eliminated as the PSorted property is preserved.

6.2.4 *Optimization Effectiveness*

Table 4 gives the performance comparisons between the original versions and the optimized ones. It shows significant read I/O reduction (47%, 35%, and 41%) for all three cases. In particular, cross-pod read data usually has a much larger reduction compared to intra-pod read data; this is because most data-shuffling stages involve a large number of machines (1,000 and 2,500) that cannot fit in a single pod except those in the first case (150).

The I/O reduction leads to the reduction of vertex numbers and vertex hours, where a vertex is a basic scheduling unit and a chain of UDFs can be continuously applied on one or more data partitions. The vertex numbers are reduced when the repartitioning step in the shuffling stages is removed, and the reduction ratio is basically proportional to the percentage of optimized shuffling stages (40%, 38%, and 82%). The reduction in vertex hour is relatively small and not proportional to the saved I/O; this is partially because the time complexity of the computation in the vertices is not necessarily linear to the input size ($O(n)$); for example, local sort takes $O(n * log(n))$, and the optimization introduces data-skew problems (described next).

Usually, significantly reduced I/O leads to much reduced end-to-end execution time (e.g., 47% vs. 40% and 35% vs. 45% for the first two cases). However, as

Case	Setting	In(GB)	Node#	E2E(min)	Vertex(hour)	Vertex#	TotalR(GB)	IPR(GB)	CPR(GB)
§ 6.2.1	original	241	150	25	35	2,974	902	102	800
	optimized	241	150	15	28	1,780	474	43	431
	reduction	-	-	40%	20%	40%	47%	59%	46%
§ 6.2.2	original	10,118	1,000	230	5,852	382,708	60,428	12,437	47,991
	optimized	10,118	1,000	127	3,350	237,751	39,080	11,658	27,422
	reduction	-	-	45%	42%	38%	35%	6%	43%
§ 6.2.3	original	241/341	2,500	96	856	352,406	14,651	2,970	11,681
	optimized	241/341	2,500	122	371	62,619	8,677	2,656	6,021
	reduction	-	-	**-27**%	57%	82%	41%	11%	48%

Table 4: Performance comparisons between unoptimized and optimized jobs. *In* indicates the input data size. *E2E* refers to the end-to-end time. *Vertex(hour)* represents the total vertex running-time across all machines. *TotalR*, *IPR*, and *CPR* report the sizes of total, intra-pod, and cross-pod read data for shuffling. The third case involves a join of two input sets, whose sizes are reported separately.

discussed before, when redefining partition keys, SUDO may bring in data-skew problems. We did observe a serious data-skew problem that causes the optimized job to be slower (-27%) than the original one in the third case, which was introduced by enforcing the records for the same sites on the same data partition before the first reduce function. The data skew problem leads to stragglers among partitions at the same computation phase and hurts the overall latency. In the future, we need more work on handling data-skew problems better and/or on the cost model for better execution plan selection.

6.3 Observations and Future Directions

During our study on the SCOPE jobs, we have also observed a set of interesting and somewhat “negative” issues that are worth further investigation. First, unlike in a typical MapReduce model, where reducers and mergers are constrained to be pass-through functions for the keys, SCOPE’s programming model is different and allows reducers and mergers to change keys. Knowing that a reducer or merger in SCOPE is a pass-through function enables more optimizations in SCOPE, but we do not count them in our study because those are specific to SCOPE. In general, it is interesting to observe that a more constrained language often leads to better automatic optimizations. A good programming language must carefully balance flexibility and optimizability.

Second, we observed that for quite a few jobs, the I/O cost is dominated by the initial “extractor” phase to load the data initially. A careful investigation shows that some of those jobs were loading more data than necessary and could benefit greatly from notions like column groups in BigTable or any column-based organizations.

Third, for some jobs, we found that the first data-shuffling stage dominates in terms of shuffling cost, as the output size of the reduce phase after that stage is significantly smaller than the input. This was one of the reasons that caused us to look at pipeline opportunities because a first phase in a job might also be optimized if it takes outputs from another job.

Finally, the rule-based deduction can be further improved. To ensure soundness, our current implementation is conservative and can be improved by making the analysis *context-sensitive* and *path-sensitive*. By being context-sensitive, SUDO’s analysis will be able to differentiate the cases where a function is invoked by different callers with different parameters; by being path-sensitive, SUDO’s analysis takes branching conditions into account. SUDO can further incorporate the value-range information to handle operators, such as `MULTIPLY`, whose functional properties depend on the value ranges of the input operands. Currently, SUDO recognizes the sign of all constant numbers automatically, but requires that developers mark the value range of an input column if it is used as a partitioning key and involved in the deduction process with value-range sensitive operators. It would be ideal not having to depend on such annotations.

7 RELATED WORK

SUDO proposes an optimization framework for data-parallel computation, as pioneered by MapReduce [15], followed by systems such as Hadoop [3] and Dryad [23]. The MapReduce model has been extended [12] with Merge to support joins and to support pipelining [13]. CIEL [30] is a universal execution engine for distributed data-flow programs. High-level and declarative programming languages, often with some SQL flavors, have been proposed. Examples include Sawzall [36], Pig [33], Hive [40], SCOPE [10], and DryadLINQ [46]. Those high-level programs are compiled into low-level execution engines such as MapReduce. Because SUDO works at the low-level Map/Reduce/Merge model, the optimizations can be applied in general to these systems: the extensive support of UDFs and heavy use of data-shuffling are common in all those proposals. The trend of embracing high-level programming languages also plays into our favor: SUDO can be integrated nicely into the compiler/optimizer, can rely on high-level semantics, and often have the extra flexibility of re-defining partition functions, as they are being generated by a compiler/optimizer. In fact, many pieces of recent work (e.g., [43, 37, 11, 21, 20]) have introduced notions of query optimizations into MapReduce and its variants.

The idea of tracking data properties has been used extensively in the database community and dates back at least to System R [38], which tracks ordering information for intermediate query results. Simmen, *et al.* [39] showed how to infer sorting properties by exploiting functional dependencies. Wang, *et al.* [41] introduced a formalism on ordering and grouping properties; it can be used to reason about ordering and grouping properties and to avoid unnecessary sorting and grouping. Neumann and Moerkotte [32, 31] also described a combined platform to optimize sorting and grouping. Zhou, *et al.* [48] are the first to take into account partitioning, in addition to sorting and grouping, in the context of the SCOPE optimizer. Such optimization heavily relies on reasoning about data properties and applies complex query transformation to choose the best execution plan in a cost-based fashion. Agarwal, *et al.* [5] collects code and data properties by piggybacking on job execution. It adapts execution plans by feeding these contextual properties to a query optimizer. The existence of UDFs prevents the system from effectively reasoning relationship between the input and the output data properties [21]. As a result, the system may miss many important optimization opportunities and end up with suboptimal query plans. SUDO builds upon the idea of reasoning about data properties, defines a set of simple and practical data-partition properties, and takes UDFs and their functional properties into account.

The need to analyze UDFs, by means of different static analysis techniques such as dataflow analysis [6, 7, 28], abstract interpretation [14], and symbolic execution [19, 22, 25], has also been recognized. Ke *et al.* [26] focuses on data statistics and computational complexity of UDFs to cope with data skew, a problem that has been also studied extensively [16, 34, 17, 9, 8]. Manimal [24] extracts relational operations such as selection, projection and data compression from UDFs through static analysis, so that traditional database optimization techniques can be applied. Manimal use the ASM bytecode manipulation library [4] to process the compiled byte-code of UDFs. Given that previous papers [26, 24] care about different properties for UDFs for different optimization properties, it is an interesting future direction to see whether a coherent UDF framework can be established to enable all those optimizations. Scooby [44] analyzed the dataflow relations of SCOPE UDFs between input and output tables, such as column independence, column equality, and non-null of column's value, by extending the Clousot analysis infrastructure [27] so that the analysis can handle .NET methods. In comparison, SUDO extends the analysis to cover functional properties such as *monotone*, *strict monotone* and *one-to-one* of UDFs, other than column equality analysis, in order to optimize data shuffling. Yuan *et al.* [45] investigated the user-defined aggregations, especially the properties of commutative and associative-decomposable to enable partial aggregation. Such properties are explicitly annotated by programmers. Jockey [18] precomputes statistics using a simulator that captures the job's complex internal dependencies, accurately and efficiently predicting the remaining run time at different resource allocations and in different stages of the job to maximize the economic utility while minimizing its impact on the rest of the cluster. Steno *et al.* [29] can translate code for declarative LINQ [2] queries to type-specialized, inlined, and loop-based imperative code that is as efficient as hand-optimized code. It applies data analysis to eliminate chains of iterators and optimizes nested queries.

8 Conclusion

Extensive use of user-defined functions and expensive data-shuffling are two defining characteristics of data-parallel computation in the map/reduce style. By defining a framework that connects functional properties, data-partition properties, and data shuffling, SUDO opens up a set of new optimization opportunities that are proven effective using the workload in a large-scale production cluster. SUDO also reflects our belief that data-parallel computation could learn not only from database systems and distributed systems, but also from programming languages and program analysis. A systematic approach that combines those perspectives in a seamless framework is likely to bring tremendous benefits.

Acknowledgement

We thank our shepherd, Srikanth Kandula, and the anonymous reviewers for their insightful comments. We are grateful to Chang Liu and Sean McDirmid for valuable feedback, and to Chao Wang, Zekan Qian, Bo Cao, and Lin Song for sharing production jobs and data.

References

[1] bddbddb. `http://bddbddb.sourceforge.net/`.

[2] LINQ. `http://msdn.microsoft.com/en-us/library/bb308959.aspx`.

[3] Hadoop. `http://lucene.apache.org/hadoop/`, June 2007.

[4] ASM. `http://asm.ow2.org/`, 2010.

[5] S. Agarwal, S. Kandula, N. Bruno, M.-C. Wu, I. Stoica, and J. Zhou. Reoptimizing data parallel computing. In *NSDI'12*, 2012.

[6] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Princiles, Techniques, and Tools*. Addison-Wesley, 1986.

[7] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann, 2001.

[8] G. Ananthanarayanan, S. Agarwal, S. Kandula, A. G. Greenberg, I. Stoica, D. Harlan, and E. Harris. Scarlett: coping with skewed content popularity in MapReduce clusters. In *EuroSys*, 2011.

[9] G. Ananthanarayanan, S. Kandula, A. G. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris. Reining in the outliers in Map-Reduce clusters using Mantri. In *OSDI*, 2010.

[10] R. Chaiken, B. Jenkins, P.-Å. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. SCOPE: easy and efficient parallel processing of massive data sets. *PVLDB*, 1(2), 2008.

[11] B. Chattopadhyay, L. Lin, W. Liu, S. Mittal, P. Aragonda, V. Lychagina, Y. Kwon, and M. Wong. Tenzing a SQL implementation on the MapReduce framework. *PVLDB*, 4(12), 2011.

[12] H. chih Yang, A. Dasdan, R.-L. Hsiao, and D. S. Parker. Map-reduce-merge: simplified relational data processing on large clusters. In *SIGMOD*, 2007.

[13] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears. MapReduce online. In *NSDI*, 2010.

[14] P. Cousot. Abstract interpretation. *ACM Comput. Surv.*, 28, June 1996.

[15] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI*, 2004.

[16] D. J. DeWitt, J. F. Naughton, D. A. Schneider, and S. Seshadri. Practical skew handling in parallel joins. In *VLDB*, 1992.

[17] B. Fan, H. Lim, D. G. Andersen, and M. Kaminsky. Small cache, big effect: Provable load balancing for randomly partitioned cluster services. In *SOCC*, 2011.

[18] A. D. Ferguson, P. Bodik, S. Kandula, E. Boutin, and R. Fonseca. Jockey: Guaranteed job latency in data parallel clusters. In *EuroSys*, 2012.

[19] T. Hansen, P. Schachte, and H. Søndergaard. State joining and splitting for the symbolic execution of binaries. In *RV*, 2009.

[20] B. He, M. Yang, Z. Guo, R. Chen, B. Su, W. Lin, and L. Zhou. Comet: batched stream processing for data intensive distributed computing. In *SoCC'10*, 2010.

[21] H. Herodotou, F. Dong, and S. Babu. MapReduce programming and cost-based optimization? Crossing this chasm with Starfish. *PVLDB*, 4(12), 2011.

[22] W. E. Howden. Experiments with a symbolic evaluation system. In *AFIPS*, 1976.

[23] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *EuroSys*, 2007.

[24] E. Jahani, M. J. Cafarella, and C. Ré. Automatic optimization for MapReduce programs. *PVLDB*, 4(6), 2011.

[25] R. H. H. Jr. Multilisp: A language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst.*, 7(4), 1985.

[26] Q. Ke, V. Prabhakaran, Y. Xie, Y. Yu, J. Wu, and J. Yang. Optimizing data partitioning for data-parallel computing. In *HotOS XIII*, 2011.

[27] F. Logozzo and M. Fähndrich. On the relative completeness of bytecode analysis versus source code analysis. In *CC*, 2008.

[28] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[29] D. G. Murray, M. Isard, and Y. Yu. Steno: automatic optimization of declarative queries. In *PLDI*, 2011.

[30] D. G. Murray, M. Schwarzkopf, C. Smowton, S. Smith, A. Madhavapeddy, and S. Hand. CIEL: a universal execution engine for distributed data-flow computing. In *NSDI*, 2011.

[31] T. Neumann and G. Moerkotte. A combined framework for grouping and order optimization. In *VLDB*, 2004.

[32] T. Neumann and G. Moerkotte. An efficient framework for order optimization. In *ICDE*, 2004.

[33] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: a not-so-foreign language for data processing. In *SIGMOD*, 2008.

[34] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker. A comparison of approaches to large-scale data analysis. In *SIGMOD*, 2009.

[35] Phoenix. `http://research.microsoft.com/en-us/collaboration/focus/cs/phoenix.aspx`, June 2008.

[36] R. Pike, S. Dorward, R. Griesemer, and S. Quinlan. Interpreting the data: Parallel analysis with Sawzall. *Scientific Programming*, 13(4), 2005.

[37] G. Sagy, D. Keren, I. Sharfman, and A. Schuster. Distributed threshold querying of general functions by a difference of monotonic representation. *PVLDB*, 4(2), 2010.

[38] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *SIGMOD*, 1979.

[39] D. E. Simmen, E. J. Shekita, and T. Malkemus. Fundamental techniques for order optimization. In *SIGMOD*, 1996.

[40] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive - a warehousing solution over a Map-Reduce framework. *PVLDB*, 2(2), 2009.

[41] X. Wang and M. Cherniack. Avoiding ordering and grouping in query processing. In *VLDB*, 2003.

[42] M. Weiser. Program slicing. In *ICSE*, 1981.

[43] S. Wu, S. Agarwal, F. Li, S. Mehrotra, and B. C. Ooi. Query optimization for massively parallel data processing. In *SoCC*, 2011.

[44] S. Xia, M. Fähndrich, and F. Logozzo. Inferring dataflow properties of user defined table processors. In *SAS*, 2009.

[45] Y. Yu, P. K. Gunda, and M. Isard. Distributed aggregation for data-parallel computing: interfaces and implementations. In *SOSP*, 2009.

[46] Y. Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P. K. Gunda, and J. Currey. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *OSDI*, 2008.

[47] J. Zhang, H. Zhou, R. Chen, X. Fan, Z. Guo, H. Lin, J. Y. Li, W. Lin, J. Zhou, and L. Zhou. Optimizing data shuffling in data-parallel computation by understanding user-defined functions. *MSR-TR-2012-28*, 2012.

[48] J. Zhou, P.-Å. Larson, and R. Chaiken. Incorporating partitioning and parallel plans into the SCOPE optimizer. In *ICDE*, 2010.

XIA: Efficient Support for Evolvable Internetworking

Dongsu Han Ashok Anand[†] Fahad Dogar Boyan Li Hyeontaek Lim
Michel Machado[*] Arvind Mukundan Wenfei Wu[†] Aditya Akella[†]
David G. Andersen John W. Byers[*] Srinivasan Seshan Peter Steenkiste

Abstract

Motivated by limitations in today's host-centric IP network, recent studies have proposed clean-slate network architectures centered around alternate first-class principals, such as content, services, or users. However, much like the host-centric IP design, elevating one principal type above others hinders communication between other principals and inhibits the network's capability to evolve. This paper presents the eXpressive Internet Architecture (XIA), an architecture with native support for multiple principals and the ability to evolve its functionality to accommodate new, as yet unforeseen, principals over time. We describe key design requirements, and demonstrate how XIA's rich addressing and forwarding semantics facilitate flexibility and evolvability, while keeping core network functions simple and efficient. We describe case studies that demonstrate key functionality XIA enables.

1 Introduction

The "narrow waist" design of the Internet has been tremendously successful, helping to create a flourishing ecosystem of applications and protocols above the waist, and diverse media, physical layers, and access technologies below. However, the Internet, almost by design, does not facilitate a clean, incremental path for the adoption of new capabilities at the waist. This shortcoming is clearly illustrated by the 15+ year deployment history of IPv6 and the difficulty of deploying primitives needed to secure the Internet. Serious barriers to evolvability arise when:

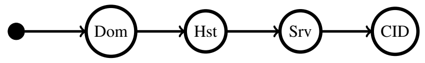
- Protocols, formats, and information must be agreed upon by a large number of independent actors in the architecture; and
- There is no built-in mechanism that supports (and embraces) incremental deployment of new functionality with minimal friction.

IP today faces both of these barriers. First, senders, receivers, and every router in between must agree on the format and meaning of the IP header. It is not possible, therefore, for a destination to switch to IPv6-based addressing and still remain reachable by unmodified senders who use IPv4. Second, today's paths to incremental deployment typically involve tunnels or overlays, which have the drawback that they *hide* the new functionality from the existing network. For example, enabling a single router in a legacy network to support some form of content-centric networking is fruitless if that traffic ends up being tunneled through the network using IPv4[1].

This paper presents a new Internet architecture, called the eXpressive Internet Architecture or XIA, that addresses these problems from the ground up. XIA maintains some features of the current Internet, such as a narrow waist that networks must support, and packet switching, but it differs from today's Internet in several areas.

The philosophy underlying the design of XIA is, simply, that we do not believe we can predict the usage models for the Internet of 50 years hence. The research community has presented compelling arguments for supporting many types of communication—content-centric networking [26, 44], service-based communication [20, 38], multicast [19], enhanced support for mobility [4, 40, 48], and so on. We believe that a new network architecture should facilitate *any or all* of these capabilities, and it must be possible to enable or disable "native" support for them as makes sense in that time and place.

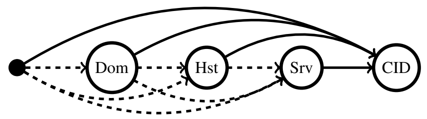
The key architectural element that XIA adds to improve evolvability is one we call expressing *intent*. XIA's addresses can simultaneously express both a "new" type of address (or addresses), and one or more backwards compatible pathways to reach that address. This notion is best explained by example: Consider the process of retrieving a particular piece of content (CID) using a network *that provides only host-to-host communication*, much like today's Internet. The source would send a packet destined to a destination network (Dom); the destination network would deliver it to a host; the host would deliver it to a process providing a service (Srv) such as HTTP; and the process would reply with the desired content, as such:



Carnegie Mellon University
[*]Department of Computer Science, Boston University
[†]University of Wisconsin-Madison

[1]Without resorting to deep packet inspection of various sorts, which itself is typically fragile.

XIA makes this path explicit in addressing, and allows flexibility in expressing it, e.g., "The source really just wants to retrieve this content, and it does not care whether it goes through Dom to get it." As a result, this process of content retrieval might be expressed in XIA by specifying the destination address as a *directed acyclic graph*, not a single address, like this:



By expressing the destination in this way, senders *give flexibility to the network to satisfy their intent*. Imagine a future network in which the destination domain supported routing directly to services [20] (instead of needing to route to a particular host). Using the address as expressed above, this hypothetical network would already have both the *information* it needs (the service ID) and *permission* to do so (the link from the source directly to the service ID). A router or other network element that does not know how to operate on, e.g., the service ID, would simply route the packet to the furthest-along node that it does know (the domain or the host in this example). Note that the sender can use the same address before and after support for service routing is introduced.

XIA terms the types of nodes in the address *principals*; examples of principals include hosts, autonomous domains (analogous to today's autonomous systems), services, content IDs, and so on. The set of principals in XIA is not fixed: hosts or applications can define new types of principals and begin using them *at any time*, without waiting for support from the network. Of course, if they want to get anything done, they must also express a way to get their work done in the current network. We believe that the ability to express not just "how to do it today", but also your underlying intent, is key to enabling future evolvability.

The second difference between XIA and today's Internet comes from a design philosophy that encourages creating principals that have *intrinsic security*: the ability for an entity to validate that it is communicating with the correct counterpart without needing access to external databases, information, or configuration. An example of an intrinsically secure address is using the hash of a public key for a host address [10]. With this mechanism, the host can prove that it sent a particular packet to any receiver who knows its host address. Intrinsic security is central to reliable sharing of information between principals and the network and to ensuring correct fulfillment of the contract between them. It can furthermore be used to bootstrap higher level security mechanisms.

We make the following contributions in this paper: We outline novel design ideas for evolution (§2), and systematically incorporate them into the eXpressive Internet Protocol (XIP) (§3). We describe optimizations for high-speed per-hop packet processing, which can achieve multi-10Gbps forwarding speed. Then, through concrete examples, we show how networks, hosts, and applications interact with each other and benefit from XIA's flexibility and evolvability (§4). Through prototype implementation and deployment, we show how applications can benefit, and demonstrate the practicality of XIP and the architecture under current and (expected) future Internet scales and technology (§5). We discuss related work (§6) and close with a conclusion and a list of new research question that XIA raises (§7).

2 Foundational Ideas of XIA

XIA is based upon three core ideas for designing an evolvable and secure Internet architecture:

1. Principal types. Applications can use one or more principal types to directly express their intent to access specific functionality. Each principal type defines its own "narrow waist", with an interface for applications and ways in which routers should process packets destined to a particular type of principal.

XIA supports an open-ended set of principal types, from the familiar (hosts), to those popular in current research (content, or services), to those that we have yet to formalize. As new principal types are introduced, applications or protocols may start to use these new principal types at any time, *even before the network has been modified to natively support the new function*. This allows incremental deployment of native network support without further change to the network endpoints, as we will explore through examples in §4.2 and §4.3.

2. Flexible addressing. XIA aims to avoid the "bootstrapping problem": why develop applications or protocols that depend on network functionality that does not yet exist, and why develop network functionality when no applications can use it? XIA provides a built-in mechanism for enabling new functions to be deployed piecewise, e.g., starting from the applications and hosts, then, if popular enough, providing gradual network support. The key challenge is: how should a legacy router in the middle of the network handle a new principal type that it does not recognize? To address this, we introduce the notion of a *fallback*. Fallbacks allow communicating parties to specify alternative action(s) if routers cannot operate upon the primary intent. We provide details in §3.2.

3. Intrinsically secure identifiers. IP is notoriously hard to secure, as network security was not a first-order consideration in its design. XIA aims to build security into the core architecture as much as possible, without impairing expressiveness. In particular, principals used in XIA source and destination addresses must be *intrinsically secure*, i.e., cryptographically derived from the associated communicating entities in a principal type-specific fashion. This allows communicating entities to more accurately ascertain the security and integrity of their transfers; for example, a publisher can attest that it delivered specific bytes to the intended recipients. While the implementation of intrinsic security is not a focus of this paper, we briefly describe the intrinsic security of our current principal types in §3.1, as well as the specification requirements for intrinsic security for new principal types.

3 XIP

XIA facilitates communication between a richer set of principals than many other architectures. We therefore split both the design and our discussion of communication within XIA into two components. First, the basic building block of per-hop communication is the core eXpressive Internet Protocol, or XIP. XIP is principal-independent, and defines an address format, packet header, and associated packet processing logic. A key element of XIP is a flexible format for specifying multiple paths to a destination principal, allowing for "fallback" or "backwards-compatible" paths that use, e.g., more traditional autonomous system and host-based communication.

The second component is the per-hop processing for each principal type. Principals are named with typed, unique eXpressive identifiers which we refer to as XIDs. In this paper, we focus on host, service, content, and administrative domain principals to provide an example of how XIA supports multiple principals. We refer to the above types as HIDs, SIDs, CIDs, and ADs, respectively. The list of principal types is extensible and more examples can be found in our prior work [8].

Our goal is for the set of mandatory and optional principals to evolve over time. We envision that an initial deployment of XIA would mandate support for ADs and HIDs, which provide global reachability for host-to-host communication—a core building block today. Support for other principal types would be optional and over time, future network architects could mandate or remove the mandate for these or other principals as the needs of the network change. Or, put more colloquially, we do not see the need for ADs and HIDs disappearing any time soon, but our own myopia should not tie the hands of future designers!

3.1 Principals

The specification of a principal *type* must define:

1. The semantics of communicating with a principal of that type.
2. A unique XID type, a method for allocating XIDs and a definition of the intrinsic security properties of any communication involving the type. These intrinsically secure XIDs should be globally unique, even if, for scalability, they are reached using hierarchical means, and they should be generated in a distributed and collision-resistant way.
3. Any principal-specific per-hop processing and routing of packets that must either be coordinated or kept consistent in a distributed fashion.

These three features together define the *principal-specific support* for a new principal type. The following paragraphs describe the administrative domain, host, service, and content principals in terms of these features.

Network and *host* principals represent autonomous routing domains and hosts that attach to the network. ADs provide hierarchy or scoping for other principals, that is, they primarily provide control over routing. Hosts have a single identifier that is constant regardless of the interface used or network that a host is attached to. ADs and HIDs are self-certifying: they are generated by hashing the public key of an autonomous domain or a host, unforgeably binding the key to the address. The format of ADs and HIDs and their intrinsic security properties are similar to those of the network and host identifiers used in AIP [10].

Services represent an application service running on one or more hosts within the network. Examples range from an SSH daemon running on a host, to a Web server, to Akamai's global content distribution service, to Google's search service. Each service will use its own application protocol, such as HTTP, for its interactions. An SID is the hash of the public key of a service. To interact with a service, an application transmits packets with the SID of the service as the destination address. Any entity communicating with an SID can verify that the service has the private key associated with the SID. This allows the communicating entity to verify the destination and bootstrap further encryption or authentication.

In today's Internet, the true endpoints of communication are typically application processes—other than, e.g., ICMP messages, very few packets are sent to an IP destination without specifying application port numbers at a higher layer. In XIA, this notion of processes as the true destination can be made explicit by specifying an SID associated with the application process (e.g., a socket) as the intent. An AD, HID pair can be used as the "legacy path" to ensure global reachability, in which case the AD forwards the packet to the host, and the host "forwards" it to the appropriate process (SID). In §4, we show that making the true process-level destination explicit facilitates transparent process migration, which is difficult in today's IP networks because the true destination is hidden as state in the receiving end-host.

Lastly, *the content principal* allows applications to express their intent to retrieve content without regard to its location. Sending a request packet to a CID initiates retrieval of the content from either a host, an in-network content cache, or other future source. CIDs are the cryptographic hash (e.g., SHA-1, RIPEMD-160) of the associated content. The self-certifying nature of this identifier allows any network element to verify that the content retrieved matches its content identifier.

3.2 XIP Addressing

Next, we introduce key concepts for XIP addresses that support the long-term evolution of principal types, the encoding mechanism for these addresses, and a representative set of addressing "styles" supported in XIP.

3.2.1 Core concepts in addressing

XIA provides native support for multiple principal types, allowing senders to express their intent by specifying a typed XID as part of the XIP destination address. However, XIA's design goal of evolvability implies that a principal type used as the intent of an XIP address may not be supported by all routers. Evolvability thus leads us to the architectural notion of **fallback**: intent that may not be globally understood *must* be expressed with an alternative backwards compatible route, such as a globally routable service or a host, that can satisfy the request corresponding to the intent. This fallback is expressed *within* an XIP address since it may be needed to reach the intended destination.

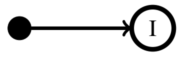
XIP addressing must also deal with the fact that not all XID types will be globally routable, for example, due to scalability issues. This problem is typically addressed through scoping based on network identifiers [10]. Since XIA supports multiple principal types, we generalize scoping by allowing the use of XID types other than ADs for scoping. For example, scaling global flat routing for CIDs may be prohibitively expensive [12, 42], and, thus, requests containing *only* a CID may not be routable. Allowing the application to refine its intent using hierarchical **scoping** using ADs, HIDs, or SIDs to help specify the CID's location can improve scalability and eliminate the need for XID-level global routing. We explore the effectiveness of using this more scalable approach in §5.3.

The drawback of scoping intent is that a narrow interpretation could limit the network's flexibility to satisfy the intent in the most efficient manner, e.g., by delivering content from the nearest cache holding a copy of the CID, rather than routing to a specific publisher. We can avoid this limitation by combining fallback and scoping, a concept we call (iterative) **refinement** of intent. When using refinement of intent, we give the XID at each scoping step the opportunity to satisfy the intent directly without having to traverse the remainder of the scoping hierarchy.

3.2.2 Addressing mechanisms

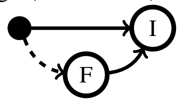
XIA's addressing scheme is a direct realization of these high-level concepts. To implement fallback, scoping, and iterative refinement, XIA uses a restricted directed acyclic graph (DAG) representation of XIDs to specify XIP addresses. A packet contains both the destination DAG and the source DAG to which a reply can be sent. Because of symmetry, we describe only the destination address.

Three basic building blocks are: intent, fallback, and scoping. XIP addresses must have a *single* intent, which can be of any XID type. The simplest XIP address has only a "dummy" source and the intent (I) as a sink:



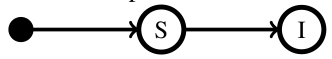
The dummy source (•) appears in all visualizations of XIP addresses to represent the conceptual source of the packet.

A fallback is represented using an additional XID (F) and a "fallback" edge (dotted line):



The fallback edge can be taken if a direct route to the intent is unavailable; we allow up to four fallbacks.

Scoping of intent is represented as:



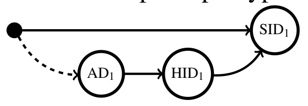
This structure means that the packet must be first routed to a scoping XID S, even if the intent is directly routable.

These building blocks are combined to form more generic DAG addresses that deliver rich semantics, implementing the high-level concepts in §3.2.1. To forward a packet, routers traverse edges in the address in order and forward using the next routable XID. Detailed behavior of packet processing is specified in §3.3.2.

3.2.3 Addressing style examples

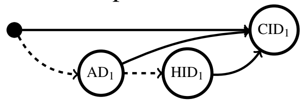
XIP's DAG addressing provides considerable flexibility. In this subsection, we present three (non-exhaustive) "styles" of how it might be used to achieve important architectural goals.

Supporting evolution: The destination address encodes a service XID as the intent, and an autonomous domain and a host are provided as a fallback path, in case routers do not understand the new principal type.



This scheme provides both fallback and scalable routing. A router outside of AD_1 that does not know how to route based on intent SID_1 directly will instead route to AD_1.

Iterative refinement: In this example, every node includes a direct edge to the intent, with fallback to domain and host-based routing. This allows iterative incremental refinement of the intent. If the CID_1 is unknown, the packet is then forwarded to AD_1. If AD_1 cannot route to the CID, it forwards the packet to HID_1.



An example of the flexibility afforded by this addressing is that an on-path content-caching router could directly reply to a CID query without forwarding the query to the content source. We term this *on-path interception*. Moreover, if technology advances to the point that content IDs became globally routable, the network and applications could benefit directly, without changes to applications.

Service binding and more: DAGs also enable application control in various contexts. In the case of legacy HTTP, while the initial packet may go to any host handling the web service, subsequent packets of the same "session" (e.g., HTTP keep-alive) *must* go to the same host. In XIA, we do so by having the initial packet destined for: $\bullet \rightarrow AD_1 \rightarrow SID_1$. A router inside AD_1 routes the request to a host that provides SID_1. The service replies with a source address bound to the host, $\bullet \rightarrow AD_1 \rightarrow HID_1 \rightarrow SID_1$, to which subsequent packets can be sent.

Other uses of DAGs are described in [9]. Some potential uses of DAG-based addresses, such as source routing, raise questions of policy and authorization that we do not explore in this paper. Here, we focus on supporting evolvability, refinement, and closely related uses such as binding and rebinding (§4).

3.3 XIP Header and Per-Hop Processing

This section describes the XIP packet format and per-hop processing that routers perform on packets. Later, in §5, we show that this design satisfies both the requirements for flexibility and efficient router processing.

3.3.1 Header format

Figure 1 shows the header format. Our header encodes a source and a destination DAG, and as a result our address is variable-length—`NumDst` and `NumSrc` indicate the size of the destination and source address. The header contains fields for version, next header, payload length, and hop limit. More details are described in [9].

Our header stores a pointer, `LastNode`, to the previously visited node in the destination address, for routers to know where to begin forwarding lookups. This makes per-hop processing more efficient by enabling routers to process a partial DAG instead of a full DAG in general.

DAGs are stored as adjacency lists. Each node in the adjacency list contains three fields: an XID Type; a 160-bit XID; and an array of the node indices that represent the node's outgoing edges in the DAG. The adjacency list format allows at most four outgoing edges per node (`Edge0...Edge3`). This choice balances: (a) the per-hop processing cost, overall header size, and simple router implementation; with (b) the desire to flexibly express many styles of addressing. However, we do not limit the degree of expressibility; one can express more outgoing edges by using a special node with a predefined `XIDType` to represent indirection.

Note that our choice of 160-bit XID adds large overhead, which could be unacceptable for bandwidth or power-limited devices. We believe, however, that common header compression techniques [30] can effectively reduce the header size substantially without much computational overhead.

3.3.2 Per-hop processing

Figure 2 depicts a simplified flow diagram for packet processing in an XIP router. The edges represent the flow of packets among processing components. Shaded elements are principal-type specific, whereas other elements are common to all principals. Our design isolates principal-type specific logic to make it easier to add support for new principals.

When a packet arrives, a router first performs source XID-specific processing based upon the XID type of the sink node of the source DAG. For example, a source DAG $\bullet \rightarrow AD_1 \rightarrow HID_1 \rightarrow CID_1$ would be passed to the CID processing module. By default, source-specific processing modules are defined as a no-op since source-specific processing is often unnecessary. In our prototype, we override this default only to define a processing module for the content principal type. A CID sink node in the source DAG represents content that is being forwarded to some destination. The prototype CID processing element opportunistically caches content to service future requests for the same CID.

The following stages of processing iteratively examine the outbound edges of the last-visited node of the DAG in priority order. We refer the node pointed by the edge in consideration as the next destination. To attempt to forward along an adjacency, the router examines the XID type of the next destination. If the router supports that principal type, it invokes a principal-specific component based on the type, and if it can forward the packet using the adjacency, it does so. If the router does not support the principal type or does not have an appropriate forwarding rule, it moves on to the next edge. This enables principal-specific customized forwarding ranging from simple route lookups to packet replication or diversion. If no outgoing edge of the last-visited node can be used for forwarding, the destination is considered unreachable and an error is generated.

3.3.3 Optimizations for high performance

The per-hop processing of XIA is more complex than that of IP, which raises obvious concerns about the performance of routers, especially in scenarios using more complex DAGs for addressing. In this section, we show that, despite those concerns, an XIP router can achieve comparable performance to IP routers by taking advantage of well-known optimizations, such as parallel processing

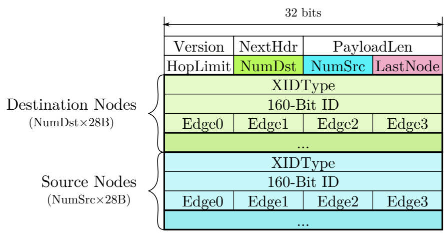


Figure 1: XIP packet header.

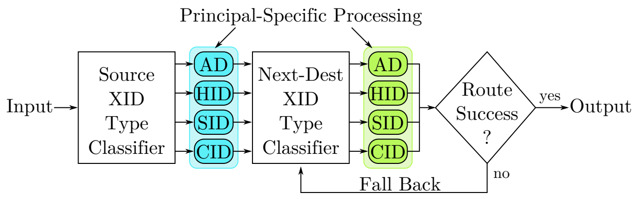

Principal-Specific Processing
Input
Source XID Type Classifier
AD
HID
SID
CID
Next-Dest XID Type Classifier
AD
HID
SID
CID
Route Success ?
yes
Output
no
Fall Back

Figure 2: Simplified diagram of an XIP router.

and fast-path evaluation.

Packet-level parallelism: By processing multiple packets concurrently, *parallel packet processing* can speed up XIP forwarding. Fortunately, in XIP, AD and HID packet processing resembles IP processing in terms of data dependencies; the forwarding path contains no per-packet state changes at all. In addition, the AD and HID lookup tables are the only shared, global data structure, and like IP forwarding tables, their update rate is relatively infrequent (once a second or so). This makes it relatively straightforward to process packets destined for ADs and HIDs in parallel. While SID and CID packet processing may have common data structures shared by pipelines, any data update can be deferred for less synchronization overhead as the processing can be opportunistic and can always fall back into AD and HID packet processing. This makes CID and SID packet processing parallelizable.

Packet-parallel processing may result in out-of-order packet delivery, which disrupts existing congestion control mechanisms in TCP/IP networks [31]. One solution is to preserve intra-flow packet ordering by serializing packets from the same flow and executing them in the same pipeline processor, or alternatively by using a reordering buffer [22, 46]. An alternative solution is to design to ensure that congestion control and reliability techniques deployed in XIP networks are more tolerant of reordering [13].

Intra-packet parallelism: As discussed earlier, a DAG may encode multiple next-hop candidates as a forwarding destination. Since the evaluation of each candidate can be done in parallel, this address structure also enables *intra-packet parallel processing*. While the different next-hops can be evaluated in parallel, the results of these lookups must be combined and only the highest priority next-hop candidate with a successful lookup should be used. Note that this synchronization stage is likely to be expensive in software implementations and this type of parallelism may be most appropriate in specialized hardware implementations.

Fast-path evaluation: Finally, the XIP design can use *fast-path* processing to speed commonly observed addresses—either as an optimization to reduce average power consumption, or to construct low cost routers that do not require the robust, worst-case performance of backbone routers. For example, our prototype leverages a look-aside cache that keeps a collision-resistant fingerprint of the destination DAG address[2] and the forwarding result (the output port and the new last-node value). When a packet's destination address matches an entry in the cache, the router simply takes the cached result and skips all other destination lookup processing. Otherwise, the router pushes the packet into the original slow-path processing path. Since the processing cost of this fast path does not depend on the address used, this performance optimization may also help conceal the impact of packet processing time variability caused by differences in DAG complexity.

In §5.1, we show that the combination of these optimizations enables XIP routers to perform almost as well as IP routers. In addition, we show that hardware implementations would be able to further close the gap between XIP and IP performance.

4 XIA Addresses in Action

We elaborate how the abstractions introduced in previous sections can be put to work to create an XIP network. The following subsections explain how addresses are created and obtained, and show how XIA's architectural components can work together to support rich applications.

4.1 Bootstrapping Addresses

We assume the existence of autonomous domains and a global inter-domain routing protocol for ADs, e.g., as discussed in [10]. We walk through how HIDs, SIDs, and CIDs join a network, and how communication occurs.

Attaching hosts to a network: Each host has a public/private key pair. As a first step, each host listens for a periodic advertisement that its AD sends out. This message contains the public key of the AD, plus possibly "well-known" services that the AD provides such as a name resolution service. Using this information, the host then sends an association packet to the AD, which will be forwarded by the AD routers to a service that can proceed

[2]The use of collision-resistant hash eliminates the need to *memcmp* potentially lengthy DAGs. The fingerprint is a collision-resistant hash on a portion of the XIP address, which consists of the last-visited node and a few next-hop nodes, effectively representing forwarding possibilities of the flow of the packets. Such an operation is shown to scale up to 222 Gb/s [41] in hardware. We assume this is implemented in the NIC. Modern network interfaces already implement hash-based flow distribution for IPv4 and IPv6 for receiver-side scaling and virtualization.

with authentication based on the respective public keys.

Advertising services and content: We have designed an XIA socket API which is described in detail in our technical report [9]. We describe here how hosts can advertise services or content using this API.

To advertise a service, a process running on a host first calls `bind()` to bind itself to a public key of the service. This binding inserts the SID (the hash of the service's public key) into the host's routing table so that the service is reachable through the host. Likewise, `putContent()` stores the CID (the hash of the content) in the host's routing table. Since at this point services and content are only locally routable, request packets will have to be scoped using the host's HID, e.g., $\bullet \rightarrow AD \rightarrow HID \rightarrow CID$, to reach the intent.

For a service or a content to be reachable more broadly, the routing information must be propagated. For example, an AD can support direct routing to services and content within its domain using an intra-domain routing protocol that propagates the SIDs and CIDs supported by each host or in-network cache to the routers. Global route propagation can be handled by an inter-domain routing protocol, subject to the AD's policies and business relationships. We leave the exact mechanism, protocol, and policy for principal-specific routing (e.g., content or service routing) as future research.

Obtaining XIP addresses: Now we look into how two parties (hosts, services or content) can obtain source and destination XIP addresses to communicate. As in today's Internet, obtaining addresses is the application's responsibility. Here, we provide a few example scenarios of how XIP addresses can be created.

Source address specifies the return address to the specific instance of the principal (i.e., a bound address). Therefore, when a principal generates a packet, the source address is generally of the form $\bullet \rightarrow AD \rightarrow HID \rightarrow XID$. The AD-prefix is given by the AD when a host attaches to the network; the HID is known by the host; and the XID is provided by the application, allowing the socket layer to create the source address. XIDs will often be ephemeral SIDs. In this case, the socket layer can automatically create an SID when `connect()` is issued to an SID socket without calling `bind()`. This is similar to the use of ephemeral ports in TCP/IP.

Destination address can be obtained in many alternative ways. One way is to use a name resolution service to resolve XIP addresses from human readable names. For example, a lookup of "Google search" in the name resolution service can return a DAG that includes the intent SID along with one or more fallback ADs that host (advertised) instances of the service (similar to today's DNS SRV records). Alternatively, a Web service can embed URLs in its pages that include an intent CID for an image or document, along with the original source

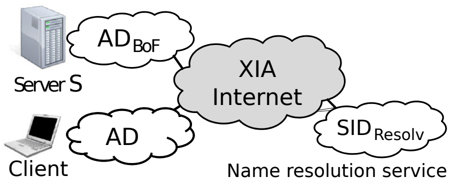


Figure 3: Bank of the Future example scenario.

($\bullet \rightarrow AD \rightarrow HID$) or content-distribution SIDs as fallbacks. This information can be in the form of a "ready-to-use" DAG, or as separate fields that can be assembled into a destination address (e.g., iterative refinement style) by the client based on local preferences. For example, the client could choose to receive content via a specific CDN based on the network interface it is using.

Note that we intentionally placed the burden of specifying fallbacks to the application layer. This is because a fallback is an authoritative location of an intent that the underlying network may not know about. Name resolution systems and other application-layer systems are more suitable to provide such information in a globally consistent manner. On the other hand, network optimizations can be applied locally in a much more dynamic fashion. Networks may choose to locally optimize for intent by locally replicating the object of intent and dynamically routing the intent.

A final point is that client ADs may have policies for what addresses are allowed. For example, it may want to specify that all packets entering or leaving the AD go through a firewall. This can be achieved by inserting an $SID_{Firewall}$ in the address, e.g., $\bullet \rightarrow AD \rightarrow SID_{Firewall} \rightarrow HID \rightarrow SID$ for a source address.

4.2 Simple Application Scenarios

In this section and the next, we use the example of online banking to walk through the lifecycle of an XIA application and its interaction with a client. Our goal is to illustrate how XIA's support for multiple principals and its addressing format give significant flexibility and control to the application.

In Figure 3, Bank of the Future (BoF) provides a secure on-line banking service hosted at a large data center on the XIA Internet. The service runs on many BoF servers and it has a public key that hashes to SID_{BoF}. We assume that all components in BoF's network natively support service and content principals. We focus on a banking interaction between a particular client host HID_C, and a particular BoF process P_S running on server HID_S.

Publishing the service: When process P_S starts on the server, it binds an SID socket to SID_{BoF} by calling `bind()` with its public/private key pair. This SID binding adds SID_{BoF} to the server's (HID_S) routing table, and the route to SID_{BoF} is advertised in the BoF network AD_{BoF}. The service also publishes the association between a human readable service name (e.g., "Bank of the Future Online") and $\bullet \rightarrow AD_{BoF} \rightarrow SID_{BoF}$ through a global name resolution service (SID_{Resolv}).

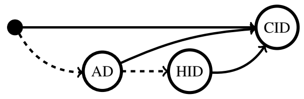
Connection initiation and binding: When a client wants to connect to the service, it first contacts the name resolution service SID_{Resolv} to obtain the service address. It then initiates a connection by sending a packet destined to $\bullet \rightarrow AD_{BoF} \rightarrow SID_{BoF}$ using the socket API. The source address is $\bullet \rightarrow AD_C \rightarrow HID_C \rightarrow SID_C$, where AD_C is the AD of the client, and SID_C is the ephemeral SID. This packet is routed to AD_{BoF} and then to an instance of SID_{BoF}. After the initial exchange, both processes will establish a session, which includes, for example, establishing a symmetric key derived from their public/private key pairs. Because of this session state, the client needs to continue to communicate with the same server, not just any server that supports SID_{BoF}. To ensure this, the client changes the destination address to $\bullet \rightarrow AD_{BoF} \rightarrow HID_S \rightarrow SID_{BoF}$, where HID_S is the server that it is currently talking to.

Content transfer: The client can now download content from the on-line banking service. For convenience, we assume that the content being transferred is a Web page. Let us consider both static (faq.html) and dynamic content (statement.html), both of which may contain static images. For *static (cacheable) content*, the SID_{BoF} service will provide the client with the CID_{faq} of the static Web page faq.html along with the CIDs of the images contained in it. The client can then issue parallel requests for those content identifiers, e.g., using $\bullet \rightarrow AD_{BoF} \rightarrow CID_{faq}$ as the destination address for the Web page. The request for *dynamic (non-cachable) content*, e.g., a list of recent bank transactions, is directly sent to SID_{BoF}.

4.3 Support for Richer Scenarios

Using the example above, we now show how XIA's addressing format can support more challenging scenarios such as evolution towards content networking, process migration, and client mobility. §5.2 provides an evaluation of these example scenarios.

Network evolution for content support: The previous section described how the client can specify static content using scoped intent. Switching to the iterative refinement style (§3):



means that routing through the specified *AD* or *HID* is optional, and opens the door for *any set of XIA routers to satisfy the client's request for static content*.

The DAG address format supports incremental deployment of content support in an XIA Internet. As a first step, BoF can deploy support for CIDs internally in its network. Even if no ISPs support CIDs, the above address will allow the delivery of the above request (using the *AD*) to the BoF network, where the intent CID can be served.

As the next step, some ISPs may incrementally deploy *on-path* caches. The above address allows them to opportunistically serve content, which may allow them to cut costs by reducing their payments to upstream service providers [6]. Over time, as support for content-centric networking expands, ISPs may make bilateral agreements to enable access to each other's (cached) content. XIA can help leverage such *off-path* caches as well; of course, it would require ISPs to exchange information about cached content and update router forwarding tables appropriately.

Process migration: XIA's addressing can also support seamless process (service) migration through re-binding. Suppose that the server process P_S migrates to another machine, namely HID_T, as part of load balancing or due to a failure; we assume that appropriate OS support for process migration is available. At the start of migration, the route to SID_{BoF} is removed from the old server and added to the new server's routing table. Before migration, the service communication was bound to $\bullet \rightarrow AD_{BoF} \rightarrow HID_S \rightarrow SID_{BoF}$. After migration, the OS notifies the ongoing connections of the new HID, and the binding is changed to $\bullet \rightarrow AD_{BoF} \rightarrow HID_T \rightarrow SID_{BoF}$ at the socket layer. Notification of the binding change propagates to the client via a packet containing the message authentication code (MAC) signed by SID_{BoF} that certifies the binding change. When the client accepts the binding change message, the client updates the socket's destination address. To minimize packet drops, in-flight packets from the client can be forwarded to the new server by putting a redirection entry to SID_{BoF} in the routing table entry of the old server.

Client mobility: The same re-binding mechanism can be used to support client mobility in a way that generalizes approaches such as TCP Migrate [40]. When a client moves and attaches to another AD, AD_{new}, the new source address of the client becomes: $\bullet \rightarrow AD_{new} \rightarrow HID_C \rightarrow SID_C$. When a rebind message arrives at the server, the server updates the binding of the client's address.

Although the locations of both the server and the client have changed in the previous two examples, the two SID end-points did not change. The intrinsic security property remains the same because it relies on the SID. Both the server and the client can verify that they are talking to the services whose public keys hash to SID_{BoF} and SID_C.

5 Evaluation

We evaluate the following three important aspects of XIA: *(i) Router processing:* Can we scale XIA's packet forwarding performance? In §5.1, we show that using techniques borrowed from fast IP forwarding, the speed of an XIA router can be made comparable to that of an IPv4 router. *(ii) Application design:* How does XIA benefit application design and performance? In §5.2, we show that use of

CPU	2x Intel Xeon L5640 2.26 GHz (12MB Cache, QPI 5.86 GT/s)
NIC	2x Intel Ethernet Server Adapter X520-T2
Motherboard	Intel Server Board S5520UR

Table 1: Router Hardware Specification

multiple principal types can simplify application design, and that applications can benefit from the type-specific in-network optimizations allowed by the architecture.

(iii) Scalable routing on XIA: How does forwarding and routing scale with the number of XIA identifiers in use? In §5.3, we show that XIA routing can scale well beyond support for today's network requirements to more extreme hypothetical scenarios.

5.1 Router Design and Performance

We first demonstrate that XIA's forwarding is fast enough for a practical deployment. We show that the packet processing speed of an XIA router is comparable to that of an IP router, and various techniques can be leveraged to further close the performance gap.

Implementation: To measure the packet processing speed, we set up a software router and a packet generator to exploit packet-level parallelism by leveraging their NIC's receiver-side-scaling (RSS) function to distribute IP and XIP packets to multiple CPU cores[3]. The implementation uses the Click modular router framework [28] for processing IP and XIP packets, and PacketShader's I/O Engine (NIC driver and library) [24] for sending and receiving packets to and from the NICs. Table 1 provides the specification of the machines.

Forwarding performance: We used a forwarding table of 351K entries based on the Route Views [35] RIB snapshot on Jan 1, 2011. IP uses this table directly; XIA pessimistically uses 351K entries for the AD forwarding table, associating each CIDR block with a distinct AD.

To measure XIA packet processing performance, we generate packets using five different DAGs for the destination address: `FB0`, `FB1`, `FB2`, `FB3`, and `VIA`. `FB0` is the baseline case where no fallback is used. `FB`*i* refers to a DAG which causes the XIA router to evaluate exactly i fallbacks and to then forward based on the $(i+1)$-th route lookup. To force this, we employ a DAG with i fallbacks: the intent identifier and the first $i-1$ fallback identifiers are not in the routing table, but the final fallback is. The last DAG, `VIA`, represents the case where an intermediate node in the DAG has been reached (e.g., arrived at the specified AD). In this case, the router must additionally update the last-visited node field in the packet header to point to the next node in the DAG before forwarding, unlike the other scenarios. Identifiers are generated based on a Pareto distribution over the set of possible destination ADs (shape parameter: 1.2) to mimic a realistic heavy-tailed distribution.

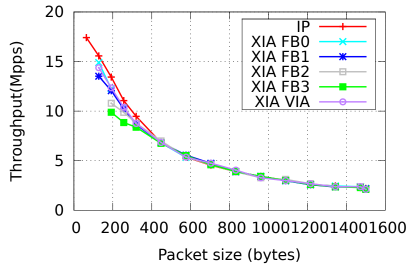
[3]To enable RSS on XIP, we prepend an IP header when generating the packet, but immediately strip the IP header after packet reception.

Figure 4 (a) and (b) respectively show the impact of varying packet size[4] on packet forwarding performance in packets and bits per second. Figure 4 (c) shows the actual goodput achieved excluding the header in each of the above experiments. The results are averaged over ten runs each lasting two minutes. Large packet forwarding is limited by I/O bandwidth, and therefore XIA and IP show little performance difference. For small packets of 192 bytes, XIA's `FB0` performance (in pps) is only 9% lower than `IP`. As more fallbacks are evaluated, performance degrades further but is still comparable to that of IPv4 (e.g., `FB3` is 26% slower than `IP`). However, the goodput is much lower due to XIA's large header size. We believe that in cases where the goodput is important, header compression techniques will be an effective solution.

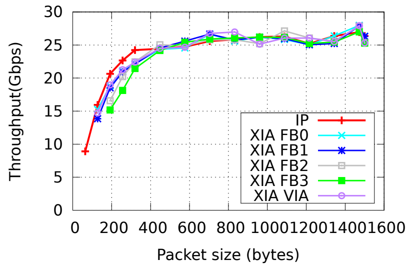
Fast-path processing: We can further reduce the gap between IP forwarding and XIP forwarding using fast-path processing techniques outlined in §3.3.3. Our fast-path implementation uses a small per-thread table, which caches route lookup results. The key used for table lookups is a collision-resistant hash of the partial DAG consisting of the last visited node and all its outbound edges. Our choice of hash domain aligns with the fact that a given router operates only on a partial DAG. We assume that the NIC hardware performs this task upon packet reception and the driver reads the hash value along with the packet (§3.3.3). We emulated this behavior in our evaluation. We generate 351K unique partial DAGs, and each thread holds 1024 entries in the lookup table. In total, the table holds 14% of these partial DAGs. Figure 5 shows the result with and without this fast-path processing for packet size of 192 bytes. Without the fast-path, the performance degrades by 19% from `FB0` to `FB3`. However, with fast-path this difference is only 7%. With a marginal performance gain of IP fast-path, the gap between `FB3` and `IP` fast-path is reduced to 10%.

Intra-packet parallelism: Note that the fast-path optimizations do not improve worst-case performance, which is often critical for high-speed routers that must forward at line speed. High-speed IP routers often rely on specialized hardware to improve worst-case performance. Although we do not have such specialized hardware for XIP, we use micro-benchmarks to estimate the performance that might be possible. The micro-benchmark results in Figure 6 show that the route lookup time is dominant and increases as more fallbacks are looked up. Fallbacks within a packet can be processed in parallel

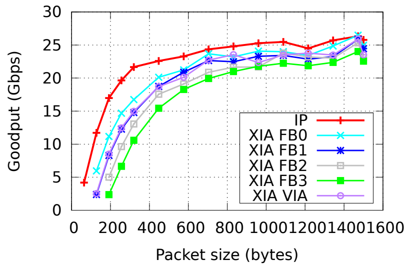
[4]We include a 14 byte MAC header in calculations of packet size and throughput. We only report the performance of packet sizes in multiples of 64 bytes because of limitations of our underlying hardware. When packet sizes do not align with the 64 byte boundary, DMA performance degrades significantly; we suspect this is triggering a known defect with our Intel hardware. This along with the additional IP and MAC header (34 bytes) and XIP's larger header size (minimum 64 bytes) resulted in a minimum packet size of 128 bytes for XIA.



(a) Throughput in Mpps



(b) Throughput in Gbps



(c) Goodput in Gbps

Figure 4: Packet forwarding performance of a software router. The forwarding table has 351 K entries.

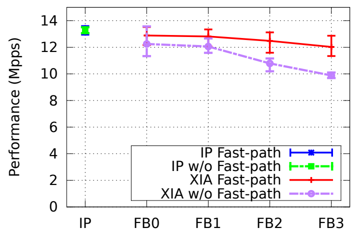


Figure 5: Fast-path processing smooths out total cost.

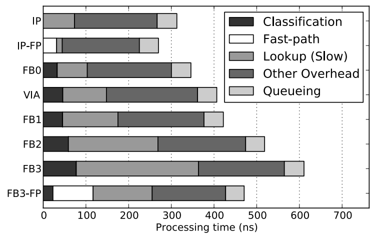


Figure 6: In-memory packet processing benchmark (no I/O).

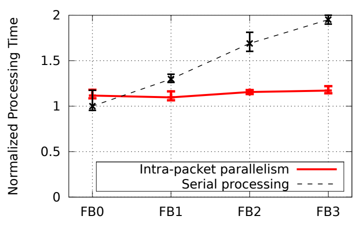


Figure 7: Parallel and serial route lookup cost.

using special-purpose hardware to further close the gap. Figure 7 shows a comparison between serial and four-way parallel lookup costs without the fast-path processing in our software router prototype, where we deliberately exclude the I/O and synchronization cost in the parallel lookup. The reason we exclude these costs is that while the overhead of intra-packet parallelism is high in our software implementation, we believe that such parallel processing overhead will be minimal in hardware router implementations or highly-parallel SIMD systems such as GPU-based software routers [24]. The figure shows that the performance gap between `FB0` and `FB3` can be eliminated with specialized parallel hardware processing.

In summary, our evaluation shows that DAG-based forwarding can be implemented efficiently enough to support high speed forwarding operations.

5.2 Application Design and Performance

We now evaluate XIP's support for applications by implementing and evaluating several application scenarios. While it is hard to quantitatively measure an architecture's support for applications, we demonstrate that XIA is able to retain many desirable features of the current Internet and subsume the benefits of other architectures [20, 26, 29]. Through this exercise, we show that XIP's flexible addressing and support for multiple principals simplifies application design, accommodates network evolution, and accelerates application performance.

Implementation: Using our XIA socket API [9], and our Click implementation of the XIP network stack, we implement in-network content support, service migration, and client mobility. To closely model realistic application behavior, we built a native XIA Web server that uses service and content principals, and a client-side HTTP-to-XIA proxy that translates HTTP requests and replies into communications based on XIP service and content principals. This proxy allows us to run an unmodified Web browser in an IP network. Implementing the proxy and server using our socket API required only 305 SLOC of Python code, and in-network content support took 742 SLOC of C++, suggesting that the principal-specific socket API and router design facilitates software design.

We now evaluate scenarios described in §4.2 and §4.3.

Content transfer and support for evolution: We created a wide-area testbed spanning two universities. The service side (CMU) operates the XIA Web server, and the client side (UWisc) performs Web browsing. The server serves dynamic and static Web pages, each of which consists of either a dynamic or static HTML file and a static image file (15 KB in each Web page). We use the addressing style described in §4.3.

Baseline content transfer: To highlight the application's use of multiple principals, we first show the baseline case where the content request takes the fallback path and the content is fetched from the origin server. Figure 8 (a) and (b) respectively show the steps and time taken to retrieve the dynamic and static Web page. When the document is dynamic (a), the server uses service-based communication to directly send the document; it also transmits a CID for the client to access the static image. In the static case (b), the service sends CIDs for both the document and the image, and the client fetches them

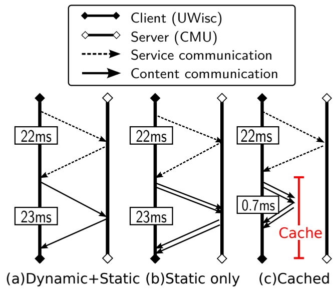


Figure 8: Content transfer.

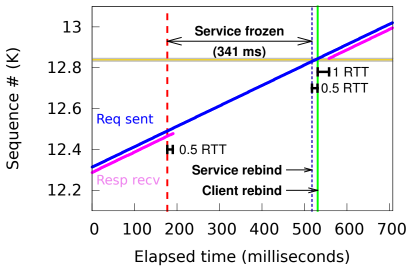


Figure 9: Service migration.

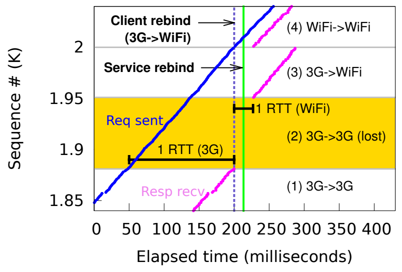


Figure 10: Client mobility.

using content communication. In both cases, it takes two round-trip times (45 ms) to retrieve the content.

Evolution: To highlight XIA's support for evolution, we consider two scenarios: 1) An in-network cache is added within the client's network without any changes to the endpoints, and 2) endpoints do not use XIA's fallback; clients first send the request to the primary intent, then redirect the intent to the original server after a timeout.

Figure 8(c) shows that content retrieval becomes faster (22.7 ms) just by adding an in-network cache without any changes (i.e., request packets in (b) and (c) are identical). This is enabled by iterative refinement-style addressing, which permits the intent to be satisfied at any step, while allowing it to fallback to the original server when it's not.

In the second scenario, the source does not use a fallback address and only uses a CID in its DAG address. The completion time becomes much worse (87 ms, not shown in the figure). The initial content request is dropped by the network because an intermediate router does not know how to route to the content. After a timeout, the application redirects the content request to the original server using the address: $\bullet \rightarrow AD_{BoF} \rightarrow HID_S \rightarrow CID$.

Service migration: As shown in §4.3, XIA supports seamless service migration with rebinding. To evaluate our service migration support, we run service SID_{BoF} on a virtual machine, which initially resides in a host machine (HID_S), and later migrates to another host (HID_T). We use KVM's live migration [3] to implement stateful migration of a running process; we move the VM along with the process running the service.

Figure 9 shows the timeline of service migration. The client makes continual requests to the service, who responds to these requests. Initially this session is bound to: $\bullet \rightarrow AD_{BoF} \rightarrow HID_S \rightarrow SID_C$. When live migration is initiated (not shown in the figure), the service continues to run on HID_S, but the underlying VM starts copying its state to the new host in the background. When most of the state is transfered to HID_T, SID_{BoF} (and the VM) is *frozen* to perform the final state transfer.

After the final state transfer, the VM and the service resume at HID_T. At this time, the service rebinds to HID_T (`Service rebind`). However, the client is still directing queries to HID_S, because it is not aware of the rebinding. The service then notifies the client of the new binding: $\bullet \rightarrow AD_{BoF} \rightarrow HID_T \rightarrow SID_C$. When the client receives this message, it rebinds to the new DAG after verification (`Client rebind`). The rebound client then starts sending subsequent requests to the new address. After one round-trip time, the client receives responses from the service. The communication is interrupted in between the rebinds. The duration of this interruption due to XIA address rebinding is minimal (the shaded region; about 1.5 RTT). The downtime due to VM migration (freezing) is much longer (341 ms) in our experiment. In a more sophisticated migration implementation, packet drops can be eliminated by buffering all packets received during the entire service interruption period at HID_S and redirecting them to the service at HID_T when it completes rebinding.

Client mobility: In this scenario, a client moves from a 3G network (RTT=150ms) to a WiFi network (RTT=25ms). The client is using a simple ping-like echo service on a server during the move. After connecting to the WiFi network, the client sends a cryptographically signed rebind message, notifying the server of the new binding: $\bullet \rightarrow AD_{new} \rightarrow HID_C \rightarrow SID_C$.

Figure 10 shows the timeline of events and the sequence numbers of packets from the echo service; blue dots indicate the time and the sequence number of the requests sent by the client, and purple dots those of the responses received by the client. Different regions (regions 1 to 4) indicate the network from which the request is sent from the client and to which the response is sent by the service. Only the packets in flight at `Client rebind` are lost (shaded area, region (2)) since they are sent to the 3G network, to which the client is no longer connected. However, after `Service rebind`, responses are sent to the WiFi network. Note that packet loss can be eliminated if the 3G network forwards these packets to the new location; this can be done by updating the routing entry for the client's HID on the 3G network. One round-trip time after `Client rebind`, the client begins receiving responses from the service. However, due to the difference in the round-trip times of the 3G and the WiFi network, packet reordering happens at the server-side, and the service responds to requests made from the 3G network prior to client rebind as well as from the WiFi network after client

	Forwarding Table	Public Key Store	Content Store
HIDs (100M)	6.25 GB	50 GB	-
SIDs (2 Billion)	125 GB	1 TB	-
CIDs (YouTube 2009 [16])	21 TB	-	168 PB
CIDs (WWW 2010 [18])	At least 227 TB	-	-

Table 2: Forwarding table size and public key store size of an AD with 100 million hosts.

rebind for a short period of time (regions (3) and (4)).

In summary, XIP's flexible addressing supports seamless network evolution and provides adequate application control, while keeping the application design simple.

5.3 Scalability

We now turn to a discussion of scalability. First, we demonstrate that XIA works as a "drop-in" replacement for IP, using AD and HID routing much as today's network uses CIDR blocks and ARP within subnets. We then examine scaling to plausible near-term scenarios, such as deploying on-path or near-path content caching, in which routers maintain large lists of content chunks stored in a nearby cache. We conclude by looking at long-term deployment ideas that would stretch the capabilities of near-term routers, such as flat routing within a large provider such as Comcast; and those requiring further advances to become plausible, such as global flat routing.

XIA at today's Internet scale. IP relies on hierarchy to scale: The "default-free" zone of BGP operates on only a few hundred thousand prefixes, not the 4 billion possible 32-bit IP addresses. XIA's AD principals can provide the same function, and we expect that at first, core routers would provide only inter-AD routing. Similar to AIP, the number of ADs should be roughly close to that of today's core BGP routing tables. §5.1 demonstrated that handling XIP forwarding with a routing table of this size is easy for a high-speed software router.

HID routing within a domain is likely to have a larger range, from a few thousand hosts, to several million. Many datacenters, for example, contain 100K or more hosts [33]. We expect that extremely large domains might split into a few smaller domains based upon geographic scope. XIP routing handles these sizes well: A 100K-entry forwarding table requires only 6.2MB in our (not memory optimized) implementation. Boosting the number of table entries from 10 K to 10 M decreases forwarding performance by only 8.3% (for Pareto-distributed flow sizes with shape=1.2) to 30.9% (for uniformly distributed flows); this slowdown is easily addressed by adding a small amount of additional lookup hardware.

While we believe naive flat forwarding will work in many networks, highly scalable systems, such as TRILL [5] and SEATTLE [27] can also be used.

Supporting tomorrow's scale. XIP provides considerable flexibility in achieving better scalability. First, XIP is not limited to a single level of hierarchy; a domain could add a second subdomain XID type beneath AD to improve its internal scalability. Second, other identifier types can be used to express hierarchy. For example, related content IDs can be grouped using a new principal (e.g., into an object ID). Doing so requires no cooperation from end-hosts: they must merely change the DAG address returned by naming to reflect the new hierarchy.

PortLand [33] suggests that a new layer-2 routing and forwarding protocol is needed to support millions of virtual machines (VMs) in data centers. In XIA, we can create a $HID_{Host} \rightarrow HID_{VM}$ hierarchy[5] that reduces the number of independently routable host identifiers and the forwarding table size by 1 to 2 orders of magnitude (we can also omit HID_{VM} by exposing all services in guest VMs to the host if the host's forwarding table is not overloaded, as in the service migration example of §5.2).

Hierarchy reduces forwarding table size at the cost of more bandwidth for headers; in XIA, this cost is modest: adding an extra XID requires 28 bytes/packet, but this addition might greatly simplify network components. Also, adding hierarchy in XIA does not hinder evolution. Using iterative refinement addressing, later networks could choose to ignore the hierarchy and route directly to the intended destination. This is the case even for hosts—were memory ever to become so cheap that global host-based routing was practical. More likely, however, this allows optimizations: A network might be willing to store a "redirect pointer" for a recently departed mobile host, similar to some optimizations proposed for Mobile IP. These pointers would operate on the host ID independent of hierarchy, but would be limited in number and duration.

On-path content caching and interception is a concrete example of opportunistic in-network optimization. A router forwards content (chunk) responses to a cache-engine, which caches the chunk. The router then intercepts the requests for chunks that are in the cache, and forwards these requests to the cache-engine, which serves the content. The forwarding table size can be small in this case, since it only contains information about the local cache (a 4 GB cache with an average object size of 7.8 KB [15] requires a forwarding table of size <32 MB.).

Hypothetical extremes. We now look at some extreme scenarios to better understand what kind of advances that are needed to make them feasible in XIA.

Some of the largest organizations have tens of millions of hosts. The largest cable operator has about 23 million customers [1], and Google is preparing to manage 10 million servers in the future [2]. YouTube (2009) has 1.8 billion videos [16] (large objects), and the World Wide Web (2010) has at least 60 billion pages (small objects). Table 2 shows the space needed to route on HIDs and SIDs

[5]For readers concerned about the performance penalty for having to go through a host, a simple filter can be inserted in modern NICs for direct access to VMs. Similar functionality for IPv4 matching is already implemented in many server-class NICs.

in an AD with 100 million hosts, and CIDs for YouTube and the Web[6]. Even though for large organizations, the HID and SID tables can fit in DRAM, its cost might be prohibitive if all devices had such a large table.

XIA does not make these extreme designs possible today; they may require non-flat identifiers or inexact routing [47], or techniques that have not yet been developed. Instead, XIP's flexible addressing makes it possible to take advantage of them if they are successfully developed in the future.

6 Related Work

Substantial prior work has examined the benefits of network architectures tailored for specific principal types. We view this work as largely complementary to ours, and we have drawn upon it in the design of individual principal types. The set of relevant architectural work is too large to cite fully, but includes proposals for content and service-centric networks, such as CCN [26], DONA [29], TRIAD [23], Serval [20], and many others.

Extensibility through indirection: One approach used in prior work to support multiple principal types devises solutions that leverage indirection, such as through name resolution or via overlays. For example, the Layered Naming Architecture [12] resolves service and data identifiers to end-point identifiers (hosts) and end-point identifiers to IP addresses. *i3* [42] uses an overlay infrastructure that mediates sender-receiver communication to provide enhanced flexibility. Like XIA, these architectures improve support for mobility, anycast, and multicast, but at the cost of additional indirection. Of course, an advantage of these approaches that leverage indirection over XIA is their ease of deployment atop today's Internet. DONA eliminates the cost of indirection by forwarding a packet in the process of name resolution. Like DONA, XIA separates the name (intent) from its locations. However, XIA differs in two key aspects: 1) XIA makes translation from name to location as part of packet forwarding and combines it with principal-specific processing. 2) DONA relies on the network for correct translation from a name to its location, but XIA relies on the backwards-compatible paths provided by the application.

Extensibility through programmability has been pursued through many efforts such as active networks [43], aiming to ease the difficulty of enhancing already-deployed networks. The biggest drawback to such approaches is resource isolation and security. In contrast, XIA does not make it easier to program new functionality into existing routers—although an active networks approach could potentially be applied in tandem.

[6] We estimate the number of SIDs by assuming that each host uses up to 20 ephemeral SIDs at a time on average. For large objects, the forwarding table is 0.0125% of the content size assuming a chunk size similar to BitTorrent. For the Web, the average object size is (7.8KB) [15]. We then double the size assuming a 50% load factor hash table for storage.

Architectures that evolve well have been a more recent focus. OPAE [21] shares our goals of supporting evolution and diversity, but their design focuses primarily on improved interfaces for inter-domain routing and applications, whereas XIA targets innovation and evolution of data plane functionality within or across domains. OPAE allows a domain to adopt any architecture, but does not specify how to do so incrementally, while XIA's fallback mechanisms allows incremental adoption of new principal types by design. Role-based architecture [14] promotes a non-layered design where a role provides a modularized functionality—similar to XIA's principal-specific processing. However, it is unclear how it allows incremental deployment of new functionality like XIA does.

Ratnasamy *et al.* propose deployable modifications to IP to enhance its evolvability [37]; but does not admit the expressiveness afforded by XIA. Others have argued that we should concede that IP (and HTTP) are here to stay, and simply evolve networks atop them [36]. However, this is not a solution in the long run; EvoArch [7] points out that merely pushing the narrow waist from layer 3 to layer 5 would result in yet another "ossified" layer.

Finally, *virtualizable networks* admit evolution by allowing many competing Internet instances to run concurrently on shared hardware [11, 39, 45]. Clark *et al.* present a compelling argument for the need to enable competition at an architectural level [17], which we internalized in our support for multiple principals. We believe that there are substantial benefits to ensuring that all applications can communicate with all other applications using a single Internet instance, but virtualizable networks offer the potential for stronger isolation properties and to support farther-reaching architectural changes than XIA (e.g., such as moving to a fully circuit-switched network). Substantial research remains in moving these architectures closer to fruition and in comparing their strengths.

Borrowed foundations: *Self-certifying identifiers* were used in many systems [25, 32]. AIP [10] used self-certifying identifiers for both network and host addresses, as XIP does, to simplify network-level security mechanisms. Several content-based networking proposals, such as DONA [29], use them to ensure the authenticity of content. Serval [20] similarly names services based upon the hash of a public key. These works demonstrate the substantial power of these intrinsically secure identifiers, which XIA generalizes to an architectural requirement.

Addressing schemes: The flexibility of DAG-style addressing has been used elsewhere, notably Slick Packets [34]. Our addressing scheme uses this concept in a new way, to provide support for network evolution.

7 Conclusion

XIA builds upon the TCP/IP stack's proven ability to accommodate technology evolution at higher and lower network layers by incorporating evolvability directly into the narrow waist of the network. XIA supports expressiveness, evolution and trustworthy operation through the use of an open-ended set of principal types, each imbued with intrinsic security. The centerpiece of our design, XIP, is a network-layer substrate that enables network innovation, and has the potential to support and amalgamate diverse sets of ideas from other clean-slate designs.

Substantial research remains to address issues such as crafting transport protocols that take advantage of content caching; devising a congestion-control mechanism that accommodates all principals; adapting or engineering suitable intra- and inter-domain routing protocols for HIDs and ADs; and incorporating trustworthy protocols that leverage intrinsic security. We view the large scope of future work as an architectural strength, showing that XIA enables a wealth of future innovations in routing, security, transport, and application design, without unduly sacrificing performance in the pursuit of flexibility.

Acknowledgments

We thank Sangjin Han for sharing his experience on router performance evaluation. We are also grateful to our shepherd Jon Crowcroft and anonymous reviewers for their feedback. This research was supported in part by the National Science Foundation under awards CNS-1040757, CNS-1040800, and CNS-1040801.

References

[1] Comcast press room - corporate overview. `http://www.comcast.com/corporate/about/pressroom/corporateoverview/corporateoverview.html`, 2011.

[2] Google: one million servers and counting. `http://www.pandia.com/sew/481-gartner.html`, 2007.

[3] KVM: Kernel based virtual machine, 2012. `http://www.linux-kvm.org/page/Main_Page`.

[4] MobilityFirst Future Internet Architecture Project. `http://mobilityfirst.winlab.rutgers.edu/`, 2010.

[5] IETF transparent interconnection of lots of links (TRILL) working group, 2012. `http://datatracker.ietf.org/wg/trill/charter/`.

[6] P. Agyapong and M. Sirbu. Economic incentives in content-centric networking: Implications for protocol design and public policy. In *Proc. Research Conference on Communications, Information and Internet Policy*, Sept. 2011.

[7] S. Akhshabi and C. Dovrolis. The evolution of layered protocol stacks leads to an hourglass-shaped architecture. In *Proc. ACM SIGCOMM*, Aug. 2011.

[8] A. Anand, F. Dogar, D. Han, B. Li, H. Lim, M. Machado, W. Wu, A. Akella, D. Andersen, J. Byers, S. Seshan, and P. Steenkiste. XIA: An architecture for an evolvable and trustworthy Internet. In *Proc. ACM Hotnets-X*, Nov. 2011.

[9] A. Anand, F. Dogar, D. Han, B. Li, H. Lim, M. Machado, W. Wu, A. Akella, D. Andersen, J. Byers, S. Seshan, and P. Steenkiste. XIA: An architecture for an evolvable and trustworthy Internet. Technical Report CMU-CS-11-100, Carnegie Mellon University, Feb. 2011.

[10] D. G. Andersen, H. Balakrishnan, N. Feamster, T. Koponen, D. Moon, and S. Shenker. Accountable Internet Protocol (AIP). In *Proc. ACM SIGCOMM*, Aug. 2008.

[11] T. Anderson, L. Peterson, S. Shenker, and J. Turner. Overcoming the Internet impasse through virtualization. *IEEE Computer*, 38, Apr. 2005.

[12] H. Balakrishnan, K. Lakshminarayanan, S. Ratnasamy, S. Shenker, I. Stoica, and M. Walfish. A layered naming architecture for the Internet. In *Proc. ACM SIGCOMM*, pages 343–352, Aug. 2004.

[13] E. Blanton and M. Allman. On making TCP more robust to packet reordering. *ACM SIGCOMM CCR*, 32:20–30, Jan. 2002.

[14] R. Braden, T. Faber, and M. Handley. From protocol stack to protocol heap: role-based architecture. *ACM SIGCOMM CCR*, 33, Jan. 2003.

[15] J. Charzinski. Traffic properties, client side cachability and CDN usage of popular web sites. In *Proc. MMB&DFT*, pages 136–150, 2010.

[16] G. Chatzopoulou, C. Sheng, and M. Faloutsos. A first step towards understanding popularity in Youtube. In *Proc. INFOCOM IEEE Conference on Computer Communications Workshops*, 2010.

[17] D. Clark, J. Wroclawski, K. Sollins, and B. Braden. Tussle in cyberspace: Defining tomorrow's Internet. In *Proc. ACM SIGCOMM*, Aug. 2002.

[18] M. de Kunder. The size of the World Wide Web. `http://www.worldwidewebsize.com/`, Jan. 2011.

[19] S. E. Deering. *Multicast Routing in a Datagram Internetwork*. PhD thesis, Stanford University, Dec. 1991.

[20] M. J. Freedman, M. Arye, P. Gopalan, S. Y. Ko, E. Nordstrom, J. Rexford, and D. Shue. Serval: An end-host stack for service-centric networking. In *Proc. USENIX NSDI*, Apr. 2012.

[21] A. Ghodsi, S. Shenker, T. Koponen, A. Singla, B. Raghavan, and J. Wilcox. Intelligent design enables architectural evolution. In *Proc. ACM Workshop on Hot Topics in Networks*, 2011.

[22] S. Govind, R. Govindarajan, and J. Kuri. Packet reordering in network processors. In *Proc. IEEE IPDPS 2007*, Mar. 2007.

[23] M. Gritter and D. R. Cheriton. TRIAD: A new next-generation Internet architecture. `http://www-dsg.stanford.edu/triad/`, July 2000.

[24] S. Han, K. Jang, K. Park, and S. Moon. PacketShader: a GPU-accelerated software router. In *Proc. ACM SIGCOMM*, Aug. 2010.

[25] HIP. Host Identity Protocol (HIP) Architecture. Interent Engineering Task Force, RFC 4423, May 2006.

[26] V. Jacobson, D. K. Smetters, J. D. Thornton, M. F. Plass, N. H. Briggs, and R. L. Braynard. Networking named content. In *Proc. ACM CoNEXT*, Dec. 2009.

[27] C. Kim, M. Caesar, and J. Rexford. Floodless in SEATTLE: A scalable ethernet architecture for large enterprises. In *Proc. ACM SIGCOMM*, Aug. 2008.

[28] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click modular router. *ACM ToCS*, 18(3):263–297, Aug. 2000.

[29] T. Koponen, M. Chawla, B.-G. Chun, A. Ermolinskiy, K. H. Kim, S. Shenker, and I. Stoica. A data-oriented (and beyond) network architecture. In *Proc. ACM SIGCOMM*, Aug. 2007.

[30] J. Lilley, J. Yang, H. Balakrishnan, and S. Seshan. A unified header compression framework for low-bandwidth links. In *Proc. ACM Mobicom*, Aug. 2000.

[31] R. Ludwig and R. H. Katz. The Eifel algorithm: making TCP robust against spurious retransmissions. *ACM SIGCOMM CCR*, 30:30–36, Jan. 2000.

[32] D. Mazières, M. Kaminsky, M. F. Kaashoek, and E. Witchel. Separating key management from file system security. In *Proc. ACM SOSP*, Dec. 1999.

[33] R. N. Mysore, A. Pamboris, N. Farrington, N. Huang, P. Miri, S. Radhakrishnan, V. Subramanya, and A. Vahdat. Portland: A scalable fault-tolerant layer2 data center network fabric. In *Proc. ACM SIGCOMM*, Aug. 2009.

[34] G. T. K. Nguyen, R. Agarwal, J. Liu, M. Caesar, B. Godfrey, and S. Shenker. Slick packets. In *Proc. SIGMETRICS*, 2011.

[35] U. of Oregon. RouteViews. `http://www.routeviews.org/`, 2012.

[36] L. Popa, A. Ghodsi, and I. Stoica. HTTP as the narrow waist of the future Internet. In *Proc. ACM Hotnets-IX*, Oct. 2010.

[37] S. Ratnasamy, S. Shenker, and S. McCanne. Towards an evolvable Internet architecture. In *Proc. ACM SIGCOMM*, Aug. 2005.

[38] U. Saif and J. Mazzola Paluska. Service-oriented network sockets. In *Proc. ACM MobiSys*, May 2003.

[39] R. Sherwood, G. Gibb, K.-K. Yap, G. Appenzeller, M. Casado, N. McKeown, and G. Parulkar. Can the production network be the testbed? In *Proc. 9th USENIX OSDI*, Oct. 2010.

[40] A. C. Snoeren and H. Balakrishnan. An end-to-end approach to host mobility. In *Proc. ACM Mobicom*, pages 155–166, Aug. 2000.

[41] M. Soliman and G. Abozaid. Performance evaluation of a high throughput crypto coprocessor using VHDL. In *Proc. ICCES*, Dec. 2010.

[42] I. Stoica, D. Adkins, S. Zhaung, S. Shenker, and S. Surana. Internet indirection infrastructure. In *Proc. ACM SIGCOMM*, pages 73–86, Aug. 2002.

[43] D. L. Tennenhouse and D. J. Wetherall. Towards an active network architecture. *ACM SIGCOMM CCR*, 26(2):5–18, Apr. 1996.

[44] D. Trossen, M. Sarela, and K. Sollins. Arguments for an information-centric internetworking architecture. *ACM SIGCOMM CCR*, 40:26–33, Apr. 2010.

[45] G. Watson, N. McKeown, and M. Casado. NetFPGA: A tool for network research and education. In *2nd workshop on Architectural Research using FPGA Platforms (WARFP)*, 2006.

[46] W. Wu, P. Demar, and M. Crawford. Sorting reordered packets with interrupt coalescing. *Computer Networks*, 53:2646–2662, Oct. 2009.

[47] M. Yu, A. Fabrikant, and J. Rexford. BUFFALO: bloom filter forwarding architecture for large organizations. In *Proc. ACM CoNEXT*, 2009.

[48] S. Q. Zhuang, K. Lai, I. Stoica, R. H. Katz, and S. Shenker. Host mobility using an Internet indirection infrastructure. In *Proc. ACM MobiSys*, May 2003.

Design and Implementation of a Consolidated Middlebox Architecture

Vyas Sekar[*], *Norbert Egi*[††], *Sylvia Ratnasamy*[†], *Michael K. Reiter*[⋆], *Guangyu Shi* [††]
[*] *Intel Labs,* [†] *UC Berkeley,* [⋆] *UNC Chapel Hill,* [††] *Huawei*

Abstract

Network deployments handle changing application, workload, and policy requirements via the deployment of specialized network appliances or "middleboxes". Today, however, middlebox platforms are expensive and closed systems, with little or no hooks for extensibility. Furthermore, they are acquired from independent vendors and deployed as standalone devices with little cohesiveness in how the ensemble of middleboxes is managed. As network requirements continue to grow in both scale and variety, this bottom-up approach puts middlebox deployments on a trajectory of growing device sprawl with corresponding escalation in capital and management costs.

To address this challenge, we present CoMb, a new architecture for middlebox deployments that systematically explores opportunities for *consolidation*, both at the level of building individual middleboxes and in managing a network of middleboxes. This paper addresses key resource management and implementation challenges that arise in exploiting the benefits of consolidation in middlebox deployments. Using a prototype implementation in Click, we show that CoMb reduces the network provisioning cost 1.8–2.5× and reduces the load imbalance in a network by 2–25×.

1 Introduction

Network appliances or "middleboxes" such as WAN optimizers, proxies, intrusion detection and prevention systems, network- and application-level firewalls, caches and load-balancers have found widespread adoption in modern networks. Several studies report on the rapid growth of this market; the market for network security appliances alone was estimated to be 6 billion dollars in 2010 and expected to rise to 10 billion in 2016 [9]. In other words, middleboxes are a critical part of today's networks and it is reasonable to expect that they will remain so for the foreseeable future.

Somewhat surprisingly then, there has been relatively little research on how middleboxes are built and deployed. Today's middlebox infrastructure has developed in a largely uncoordinated manner—a new form of middlebox typically emerging as a one-off solution to a specific need, "patched" into the infrastructure through adhoc and often manual techniques.

This bottom-up approach leads to two serious forms of inefficiency. The first is inefficiency in the use of infrastructure hardware resources. Middlebox applications are typically resource intensive and each middlebox is independently provisioned for peak load. Today, because each middlebox is deployed as a separate device, these resources cannot be amortized across applications even though their workloads offer natural opportunities to do so. (We elaborate on this in §3). Second, a bottom-up approach leads to inefficiencies in *management*. Today, each type of middlebox application has its own custom configuration interface, with no hooks or tools that offer network administrators a unified view by which to manage middleboxes across the network.

As middlebox deployments continue to grow in both scale and variety, these inefficiencies are increasingly problematic—middlebox infrastructure is on a trajectory of growing device sprawl with corresponding escalation in capital and management costs. In §2, we present measured and anecdotal evidence that highlights these concerns in a real-world enterprise environment.

This paper presents *CoMb*,[1] a top-down design for middlebox infrastructure that aims to tackle the above inefficiencies. The key observation in CoMb is that the above inefficiencies arise because middleboxes are built and managed as *standalone* devices. To address this, we turn to the age-old idea of *consolidation* and systematically re-architect middlebox infrastructure to exploit opportunities for consolidation. Corresponding to the inefficiencies, CoMb targets consolidation at two levels:

1. *Individual middleboxes:* In contrast to standalone, specialized middleboxes, CoMb decouples the hardware and software, and thus enables software-based implementations of middlebox applications to run on a consolidated hardware platform.[2]
2. *Managing an ensemble of middleboxes:* CoMb consolidates the management of different middlebox applications/devices into a single (logically) centralized controller that takes a unified, network-wide view—generating configurations and accounting for policy requirements across all traffic, all applications, and all network locations. This is in contrast to today's approach where each middlebox application and/or device is managed independently.

In a general context, the above strategies are not new. There is a growing literature on centralized network management (e.g., [21, 31]), and consolidation is commonly used in data centers. To our knowledge, however,

[1]The name CoMb captures our goal of *Co*nsolidating *M*iddle*b*oxes.
[2]As we discuss in §4, this hardware platform can comprise both general-purpose and specialized components.

there has been no work on quantifying the benefits of consolidation for middlebox infrastructure, nor any in-depth attempt to re-architect middleboxes (at both the device- and network-level) to exploit consolidation.

Consolidation effectively "de-specializes" middlebox infrastructure since it forces greater modularity and extensibility. Typically, moving from a specialized architecture to one that is more general results in less, not more, efficient resource utilization. We show, however, that consolidation creates *new opportunities* for efficient use of hardware resources. For example, within an individual box, we can reduce resource requirements by leveraging previously unexploitable opportunities to *multiplex* hardware resources and *reuse* processing modules across different applications. Similarly, consolidating middlebox management into a network-wide view exposes the option of *spatially* distributing middlebox processing to use resources at different locations.

However, the benefits of consolidation come with challenges. The primary challenge is that of *resource management* since middlebox hardware resources are now shared across multiple heterogeneous applications and across the network. We thus need a resource management solution that matches demands (*i.e.*, what subset of traffic needs to be processed by each application, what resources are required by different applications) to resource availability (*e.g.*, CPU cycles and memory at various network locations). In §4 and §5, we develop a hierarchical strategy that operates at two levels—network-wide and within an individual box—to ensure the network's traffic processing demands are met while minimizing resource consumption.

We prototype a CoMb network controller leveraging off-the-shelf optimization solvers. We build a prototype CoMb middlebox platform using Click [30] running on general-purpose server hardware. As test applications we use: (i) existing software implementations of middlebox applications (that we use with minimal modification) and (ii) applications that we implement using a modular datapath. (The latter were developed to capture the benefits of processing reuse). Using our prototype and trace-driven evaluations, we show that:

- At a network-wide level, CoMb reduces aggregate resource consumption by a factor 1.8–2.5$\times$ and reduces the maximum per-box load by a factor 2–25$\times$.
- Within an individual box, CoMb imposes little or minimal overhead for existing middlebox applications. In the worst case, we record a 0.7% performance drop relative to running the same applications independently on dedicated hardware.

Roadmap: In the rest of the paper, we begin with a motivating case study in §2. §3 highlights the new efficiency opportunities with CoMb and §4 describes the design of the network controller. We describe the design of each CoMb box in §5 and our prototype implementation in §6. We evaluate the potential benefits and overheads with CoMb in §7. We discuss concerns about isolation and deployment in §8. We present related work in §9 before concluding in §10.

Appliance type	Number
Firewalls	166
NIDS	127
Conferencing/Media gateways	110
Load balancers	67
Proxy caches	66
VPN devices	45
WAN optimizers	44
Voice gateways	11
Middleboxes total	636
Routers	≈ 900

Table 1: *Devices in the enterprise network*

2 Motivation

We begin with anecdotal evidence in support of our claim that middlebox deployments constitute a vital component in modern networks and the challenges that arise therein. Our observations are based on a study of middlebox deployment in a large enterprise network and discussions with the enterprise's administrators. The enterprise spans tens of sites and serves more than 80K users [36].

Table 1 summarizes the types and numbers of different middleboxes in the enterprise and shows that the total number of middleboxes is comparable to the number of routers! Middleboxes are thus a vital portion of the enterprise's network infrastructure. We further see a large diversity in the type of middleboxes; other studies suggest similar diversity in ISPs and datacenters as well [13, 26].

The administrators indicated that middleboxes represent a significant fraction of their (network) capital expenses and expressed the belief that processing complexity contributes to high capital costs. They expressed further concern over anticipated mounting costs. Two nuggets emerged from their concerns. First, they revealed that each class of middleboxes is currently managed by a *dedicated team* of administrators. This is in part because the enterprise uses different vendors for each application in Table 1; the understanding required to manage and configure each class of middlebox leads to inefficient use of administrator expertise and significant operational expenses. The lack of high-level configuration interfaces further exacerbates the problem. For example, significant effort was required to manually tune what subset of traffic should be directed to the WAN optimizers to balance the tradeoff between the bandwidth savings and appliance load. The second nugget of interest was their concern that the advent of consumer devices (e.g., smartphones, tablets) is likely to increase the need for in-network capabilities [9]. The lack of *extensibility* in middleboxes today inevitably leads to further appliance sprawl, with associated increases in capital and operating expenses.

Despite these concerns, administrators reiterated the value they find in such appliances, particularly in supporting new applications (e.g., teleconferencing), increasing security (e.g., IDS), and improving performance (e.g., WAN optimizers).

3 CoMb: Overview and Opportunities

The previous discussion shows that even though middleboxes are a critical part of the network infrastructure, they remain *expensive*, *closed* platforms that are *difficult to extend*, and *difficult to manage*. This motivates us to rethink how middleboxes are designed and managed. We envision an alternative architecture, called CoMb, wherein **software-centric** implementations of middlebox applications are **consolidated** to run on a shared hardware platform, and managed in a **logically centralized** manner.

The qualitative benefits of this proposed architecture are easy to see. Software-based solutions reduce the cost and development cycles to build and deploy new middlebox applications (as independently argued in parallel work [12]). Consolidating multiple applications on the same physical platform reduces device sprawl; we already see early commercial offerings in this regard (e.g., [3]). Finally, the use of centralization to simplify network management is also well known [31, 21, 15].

While the qualitative appeal is evident, there are practical concerns with respect to efficiency. Typically, moving from a specialized architecture to one that is more general and extensible results in less efficient resource utilization. However, as we show next, CoMb introduces *new* efficiency opportunities that do not arise with today's middlebox deployments.

3.1 Application multiplexing

Consider a WAN optimizer and IDS running at an enterprise site. The former optimizes file transfers between two enterprise sites and may see peak load at night when system backups are run. In contrast, the IDS may see peak load during the day because it monitors users' web traffic. Suppose the volumes of traffic processed by the WAN optimizer and IDS at two time instants t_1, t_2 are $10, 50$ packets and $50, 10$ packets respectively. Today each hardware device must be provisioned to handle its peak load resulting in a total provisioning cost corresponding to $2 * \max\{10, 50\} = 100$ packets. A CoMb box, running both a WAN optimizer and the IDS on the same hardware can flexibly allocate resources as the load varies. Thus, it needs to be provisioned to handle the peak *total* load of 60 packets or 40% fewer resources.

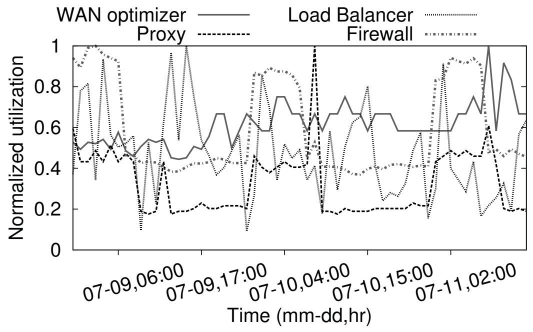


Figure 1: *Middlebox utilization peak at different times*

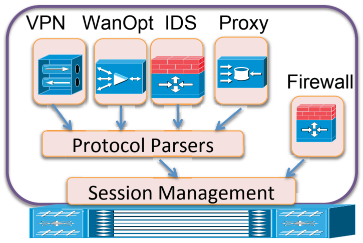


Figure 2: *Reusing lower-layer software modules across middlebox applications*

Figure 1 shows a time series of the utilization of four middleboxes at an enterprise site, each normalized by its maximum observed value. Let $NormUtil^t_{app}$ denote the normalized utilization of the device *app* at time t. Now, to quantify the benefits of multiplexing, we compare the *sum of the peak* utilizations $\sum_{app} \max_t \{NormUtil^t_{app}\} = 4$ and the *peak total* utilization $\max_t \{\sum_{app} NormUtil^t_{app}\} = 2.86$. For the workload shown in Figure 1, multiplexing requires $\frac{4-2.86}{4} = 28\%$ fewer resources.

3.2 Reusing software elements

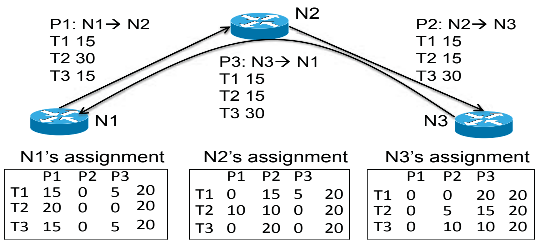
Each middlebox typically needs low-level modules for packet capture, parsing headers, reconstructing session state, parsing application-layer protocols and so on. If the same traffic is processed by many applications—e.g., HTTP traffic is processed by an IDS, proxy, and an application firewall—each appliance has to repeat these common actions for *every packet*. When these applications run on a consolidated platform, we can potentially *reuse* these basic modules (Figure 2).

Consider an IDS and proxy. Both need to reconstruct session- and application-layer state before running higher-level actions. Suppose each device needs 1 unit of processing per packet. For the purposes of this example, let us assume that these common tasks contribute 50% of the overall processing cost. Both appliances process HTTP traffic, but may also process traffic unique to each context; e.g., IDS processes UDP traffic which the proxy ignores. Suppose there are 10 UDP packets and 45 HTTP packets. The total resource requirement is $(IDS = 10 + 45) + (Proxy = 45) = 100$ units. The setup in Figure 2 with reusable modules avoids duplicating the common tasks for HTTP traffic and needs $45 * 0.5 = 22.5$ fewer resources.

As this example shows, this reduction depends on the traffic overlap across applications and the contribution of the reusable modules. To measure the overlap, we obtain configurations for Bro [32] and Snort[3] and the configuration for a WAN optimizer. Then, using flow-level traces from Internet2, we find that the traffic overlap between applications is typically 64–99% [36]. Our benchmarks in §7.1 show that common modules contribute 26–88% across applications.

3.3 Spatial distribution

Consider the topology in Figure 3 with three nodes N1–N3 and three end-to-end paths P1–P3. The traffic on these paths peaks to 30 packets at different times as shown. Suppose we want all traffic to be monitored by IDSes. Today's default deployment is an IDS at each *ingress* N1, N2, and N3 for monitoring traffic on P1, P2, and P3 respectively. Each such IDS needs to be provisioned to handle the peak volume of 30 units with a total network-wide cost of 90 units.



	P1	P2	P3	
T1	15	0	5	20
T2	20	0	0	20
T3	15	0	5	20

	P1	P2	P3	
T1	0	15	5	20
T2	10	10	0	20
T3	0	20	0	20

	P1	P2	P3	
T1	0	0	20	20
T2	0	5	15	20
T3	0	10	10	20

Figure 3: *Spatial distribution as traffic changes*

With a centralized network-wide view, however, we can *spatially distribute* the IDS responsibilities. That is, each IDS at N1–N3 processes a fraction of the traffic on the paths traversing the node (e.g., [37]). For example, at time T1, N1 uses 15 units for P1 and 5 for P3; N2 uses 15 units for P2 and 5 P3; and N3 devotes all 20 units to P3. We can generate similar configurations for the other times as shown in Figure 3. Thus, distribution reduces the total provisioning cost $\frac{90-60}{90} = 33\%$ compared to an ingress-only deployment. Note that this is orthogonal to application multiplexing and software reuse.

Using time-varying traffic matrices from Internet and the Enterprise network, we find that spatial distribution can provide 33–55% savings in practice.

3.4 CoMb Overview

Building on these opportunities, we envision the architecture in Figure 4. Each CoMb box runs multiple software-based applications (e.g., IDS, Proxy). These

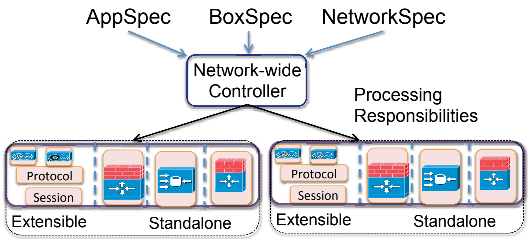


Figure 4: *The network controller assigns processing responsibilities to each CoMb box.*

applications can be obtained from independent vendors and could differ in their software architectures (e.g., standalone vs. modular). CoMb's *network controller* assigns processing responsibilities across the network. Each CoMb middlebox receives this configuration and allocates hardware resources to the different applications.

4 CoMb Network Controller

In this section, we describe the design of CoMb's network controller and the management problem it solves to assign network-wide middlebox responsibilities.

4.1 Input Parameters

We begin by describing the three high-level inputs that the network controller needs.

- *AppSpec:* For each application m (e.g., IDS, proxy, firewall), the AppSpec specifies: (1) T^m, the traffic that m needs to run on (e.g., what ports and prefixes), and (2) policy constraints that the administrator wants to enforce across different middlebox instances. These constraints specify constraints on the processing *order* for each packet [25]. For example, all web traffic goes through a firewall, then an IDS, and finally a web proxy. Most middlebox applications today operate at a session-level granularity and we assume each m operates at this granularity.[4]
- *NetworkSpec:* This has two components: (1) a description of end-to-end routing paths and the location of the middlebox nodes on each path and, (2) a partition $T = \{T_c\}_c$ of all traffic into *classes*. Each class T_c can be specified with a high-level description of the form "port-80 sessions initiated by hosts at ingress A to servers in egress B" or described by more precise *traffic filters* defined on the IP 5-tuple (e.g., srcIP=10.1.∗.∗, dstIP=10.2.∗.∗, dstport=80, srcport=∗). For brevity, we assume each class T_c has a single end-to-end path with the forward and reverse flows within a session following the same path (in opposite directions). Each application m subscribes to one or more of these traffic classes; i.e., $T^m \in 2^T$.

[3]`www.snort.org`

[4]It is easy to extend to applications that operate at per-packet or per-flow granularity; we do not discuss this for brevity.

- *BoxSpec:* This captures the hardware capabilities of the middlebox hardware: $Prov_{n,r}$ is the amount of resource r (e.g., CPU, memory) that node n is *provisioned*, in units suitable for that resource. Each platform may optionally support specialized accelerators (e.g., GPU units or crypto co-processors).

  Given the hardware configurations, we also need the (expected) per-session *resource footprint*, on the resource r, of running the application m. Each m may have some affinity for *hardware accelerators*; e.g., some IDSes use hardware-based DPI. These requirements may be strict (i.e., the application only works with hardware support) or opportunistic (i.e., offload for better performance). Now, the middlebox hardware at each node n may or may not have such accelerators. Thus, we use generalized resource footprints $F_{m,r,n}$ that depend on the specific middlebox node to account for the presence or absence of such hardware accelerators. Specifically, the footprint will be higher on a node without an optional hardware accelerator and the application needs to emulate this feature in software.[5]

In practice, these inputs are already available or easy to obtain. The *NetworkSpec* for routing and traffic information is collected for other network management applications [20]. The traffic classes and policy constraints in *AppSpec* and the hardware capacities $Prov_{n,r}$s are known; we simply require that these be made available to the network controller. The only component that imposes new effort is the set of $F_{m,r,n}$ values in *BoxSpec*. These can be obtained by running *offline* benchmarks similar to §7; even this effort is required infrequently (e.g., only after hardware upgrades).

4.2 Problem Formulation

Given these inputs, the controller's goal is to assign processing responsibilities to middleboxes across the network. There are three high-level types of constraints that this assignment should satisfy:

1. *Processing coverage:* We need to ensure that each session of interest to middlebox application m will be processed by an instance of m along that session's routing path.
2. *Policy dependencies:* For each session, we have to respect the policy ordering constraints (e.g., firewall before proxy) across middlebox applications that need to process this session.
3. *Reuse dependencies:* We need to model the potential for reusing common actions across middlebox applications (e.g., session reassembly in Figure 2).

[5]To capture strict requirements, where some application cannot run without a hardware accelerator, we set the F values for nodes without this accelerator to ∞ or some large value.

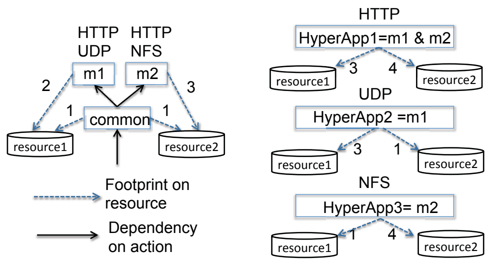


Figure 5: *Each hyperapp is a single logical task whose footprint is equivalent to taking the logical union of its constituent actions.*

Given these constraints, we can consider different management objectives: (1) minimizing the cost to *provision* the network, $\min \sum_{n,r} Prov_{n,r}$, to handle a given set of traffic patterns, or (2) *load balancing* to minimize the maximum load across the network, $\min \max_{n,r}\{load_{n,r}\}$, for the current workload and provisioning regime.

Unfortunately, the reuse and policy dependencies make this optimization problem intractable.[6] So, we consider a practical, but constrained, operating model that eliminates the need to explicitly capture these dependencies. The main idea is that *all* applications pertaining to a given session run on the same node. That is, if some session i needs to be processed by applications m_1 and m_2 (and nothing else), then we force both m_1 and m_2 to process session i on the same node. As an example, let us consider two applications: m_1 (say IDS) processes HTTP and UDP traffic and m_2 (say WAN-optimizer) processes HTTP and NFS traffic. Now, consider a HTTP session i. In theory, we could run m_1 on node n_1 and m_2 on node n_2 for this session i. Our model constrains both m_1 and m_2 for session i run on n_1. Note that we can still assign different sessions to other nodes. That is, for a different HTTP session i', m_1 and m_2 could run on n_2.

Having chosen this operational model, for each traffic class c we identify the *exact sequence* of applications that need to process sessions belonging to c. We call each such sequence a *hyperapp*. Formally, if h_c is the hyperapp for the traffic class c, then $\forall m : T_c \in T^m \Leftrightarrow m \in h_c$. (Note that different classes could have the same hyperapp.) Figure 5 shows the three hyperapps for the previous example: one for HTTP traffic (processed by both m_1 and m_2), and one each for UDP and NFS traffic processed by either m_1 or m_2 but not both. Each hyperapp also statically defines the *policy order* across its constituent applications.

This model has three practical advantages. First, it eliminates the need to capture the individual actions within a middlebox application and their reuse dependencies. Similar to the per-session resource footprint $F_{m,r,n}$ of the middlebox application m on resource r, we

[6]We show the precise formulation in a technical report [35].

can define the per-session hyperapp-footprint of the hyperapp h on resource r as $F_{h,r,n}$. This implicitly accounts for the common actions across applications within h. Note that the right hand side of Figure 5 does not show the common action; instead, we include the costs of the common action when computing the F values for each hyperapp. Identifying the hyperapps and their F values requires a pre-processing step that takes exponential time as a function of the number of applications. Fortunately, this is a one-time task and there are only a handful (< 10) of applications in practice.

Second, it obviates the need to explicitly model the ordering constraints across applications. Because all applications relevant to a session run on the same node, enforcing policy ordering can be implemented as a local scheduling decision on each CoMb box (§5).

Third, it simplifies the traffic model. Instead of considering the coverage on a per-session basis, we consider the *total volume* of traffic in each class. Thus, we can consider the management problem in terms of deciding the *fraction of traffic* belonging to the class c that each node n has to process (i.e., run the hyperapp h_c). Let $d_{c,n}$ denote this fraction and let $|T_c|$ denote the *volume* of traffic for class c.

The optimization problem can be expressed by the linear program shown in Eq(1)—Eq(4). (For brevity, we show only the load balancing objective.) Eq(2) models the stress or load on each resource at each node in terms of the aggregate processing costs (i.e., product of the traffic volume and the footprints) assigned to this node. Here, $n \in_{path} c$ denotes that node n is on the routing path for the traffic in T_c. Eq(3) simply specifies a coverage constraint so that the fractional responsibilities across the nodes on the path for each class c add up to 1.

$$\text{Minimize} \max_{r,n}\{load_{n,r}\}, \quad \text{subject to} \tag{1}$$

$$\forall n, r : load_{n,r} = \sum_{c:n \in_{path} c} \frac{d_{c,n}|T_c|F_{h_c,r,n}}{Prov_{n,r}} \tag{2}$$

$$\forall c : \sum_{n \in_{path} c} d_{c,n} = 1 \tag{3}$$

$$\forall c, n : \ 0 \le d_{c,n} \le 1 \tag{4}$$

The controller solves this optimization to find the optimal set of $d_{c,n}$ values specifying the per-class responsibilities of each middlebox node. Then it maps these values into device-level configurations per middlebox. We defer a discussion of the mapping step to §6.

5 CoMb Single-box Design

We now turn to the design of a single CoMb box. As described in the previous section, the output of the network controller is an assignment of processing responsibilities to each CoMb box. This assignment specifies:

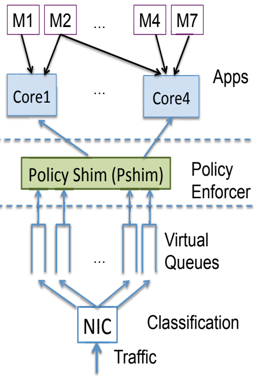


Figure 6: *Logical view of a CoMb box with three layers: classification, policy enforcement, and middlebox applications*

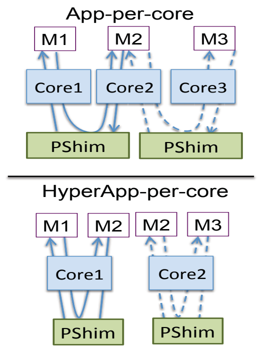


Figure 7: *An example with two hyperapps: m_1 followed by m_2 and m_2 followed by m_3. The hyperapp-per-core architecture clones m_2.*

- a set of (traffic class, fraction) pairs $\{(T_c, d_{c,n})\}$ that describes what traffic (type and volume) each CoMb box needs to process.
- the hyperapp h_c associated with each traffic class T_c, where each hyperapp is an ordered set of one or more middlebox applications.

We start with our overall system architecture and then describe how we parallelize this architecture over a CoMb box's hardware resources.

5.1 System Architecture

At a high level, packet processing within a CoMb box comprises three logical stages as shown in Figure 6. An incoming packet must first be ***classified***, to identify what traffic class T_c it belongs to. Next, the packet is handed to a ***policy enforcement*** layer responsible for steering the packet between the different applications corresponding to the packet's traffic class, in the appropriate order. Finally, the packet is processed by the appropriate ***middlebox application(s)***. Of these, classification and policy enforcement are a consequence of our consolidated design and hence we aim to make these as lightweight as possible.We elaborate on the role and design options for each stage next.

Classification: The CoMb box receives a stream of undifferentiated packets. Since different packets may be processed by different applications, we must first identify what traffic class a packet belongs to. There are two broad design options here. The first is to do the classification in hardware. Many commercial appliances rely on custom NICs for sophisticated high-speed classification and even commodity server NICs today support such capabilities [4]. A common feature across these NICs is that they support a large number of hardware queues (on the NIC itself) and can be configured to triage incoming packets into these queues using certain functions (typically exact-, prefix- and range-matches) defined on the packet headers. The second option is software-based classification—incoming packets are classified entirely in software and placed into one of multiple software queues.

The tradeoff between the two options is one of efficiency vs. flexibility. Software classification is fully general and programmable but consumes significant processing resources; e.g., Ma et al. report general software-based classification at 15 Gbps (comparable to a commodity NIC) on a 8-core Intel Xeon X5550 server [28].

Our current implementation assumes hardware classification. From an architectural standpoint, however, the two options are equivalent in the abstraction they expose to the higher layers: multiple (hardware or software) queues with packets from a traffic class T_c mapped to a dedicated queue.

We assume that the classifier has at least as many queues as there are hyperapps. This is reasonable since existing commodity NICs already have 128/256 queues per interface, specialized NICs even more, and software-based classification can define as many as needed. For example, with 6 applications, the *worst-case* number of hyperapps and virtual queues is $2^6 = 64$, which today's commodity NICs can support.

Policy Enforcer: The job of the policy enforcement layer is to 'steer' a packet in the correct order between the different applications associated with the packet's hyperapp. We need such a layer because the applications on a CoMb box could come from independent vendors and we want to run applications such that they are oblivious to our consolidation. Hence, for a hyperapp comprised of (say) IDS followed by Proxy, the IDS application would not know to send the packet to the Proxy for further processing. Since we do not want to modify applications, we introduce a lightweight *policy shim (pshim)* layer.

We leverage the above classification architecture to design a very lightweight policy enforcement layer. We simply associate a separate instance of a pshim with each output queue of the classifier. Since each queue only receives packets for a single hyperapp, the associated pshim knows that *all* the packets it receives are to be "routed" through the identical sequence of applications.

Thus, beyond retaining the sequence of applications for its associated hyperapp/traffic-class, the pshim does not require any complex annotation of packets or keep per-session state. In fact, if the hyperapp consists of a single application, the pshim is essentially a NOP.

Applications: Our design supports two application software architectures: (1) standalone software processes (that run with little or no modification) and (2) applications built atop an 'enhanced' network stack with reusable software modules for common tasks such as session reconstruction and protocol parsing. We currently assume that applications using custom accelerators access these using their own libraries.

5.2 Parallelization on a CoMb box

We assume a CoMb box offers a number of parallel computation cores—such parallelism exists in general-purpose servers (*e.g.*, our prototype server uses 8 Intel Xeon 'Westmere' cores) and is even more prevalent in specialized networking hardware (*e.g.*, Cisco's QuantumFlow packet processor offers 40 Tensilica cores). We now describe how we parallelize the functional layers described earlier on this underlying hardware.

Parallelizing the classifier: Since we assumed hardware classification, our classifier runs on the NIC and does not require parallelization across cores. We refer the reader to [28] for a discussion of how a software-based classifier might run on a multi-core system.

Parallelizing a single hyperapp: Recall that a hyperapp is a sequence of middlebox applications that need to process a packet. There are two options to map a logical hyperapp to the parallel hardware (Figure 7):

1. *App-per-core:* Each application in the hyperapp runs on a separate core and the pshim steers each packet between cores.
2. *hyperapp-per-core:* All applications belonging to the hyperapp run on the same core. Hence, a given application module is cloned with as many instances as the number of hyperapps in which it appears.

The advantage of the hyperapp-per-core approach is that a packet is processed in its entirety on a single core, avoiding the overhead of inter-core communication and cache invalidations that may arise as shared state is accessed by multiple cores. (This overhead occurs more frequently for applications built to reuse processing modules in a common stack.) The disadvantage of the hyperapp-per-core relative to the app-per-core, is that it could incur overhead due to context switches and potential contention over shared resources (*e.g.*, data and instruction caches) on a single core. Which way the scale tips depends on the overheads associated with inter-core communication, context switches, *etc.* which vary across hardware platforms.

We ran several tests across different hyperapp scenarios on our prototype server (§7) and found that the hyperapp-per-core approach offered superior or comparable performance [35]. These results are also consistent with independent measurements on software routers [17]. In light of these results, we choose the hyperapp-per-core model because it simplifies how we parallelize the pshim (see below) and ensures core-local access to reusable modules and data structures.

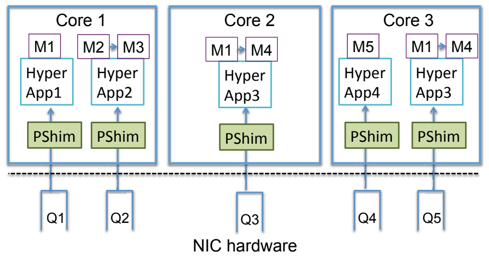


Figure 8: *CoMb box: Putting the pieces together*

Parallelizing the pshim layer: Recall that we have a separate instance of a pshim for each hyperapp. Given the hyperapp-per-core approach, parallelizing the pshim layer is easy. We simply run the pshim instance on the same core as its associated hyperapp.

Parallelizing multiple hyperapps: We are left with one outstanding question. Given multiple hyperapps, how many cores, or fraction of a core, should we assign each hyperapp? For example, the total workload for some hyperapp might exceed the processing capacity of a single core. At the same time, we also want to avoid a skewed allocation across cores. This hyperapp-to-core mapping problem can be expressed as a simple linear program that assigns a fraction of the traffic for each hyperapp h to each core. (We do not show it for brevity; please refer our technical report [35].) In practice, this calculation need not occur at the CoMb box as the controller can also run this optimization and push the resulting configuration.

5.3 Recap and Discussion

Combining the previous design decisions brings us to the design in Figure 8. We see that:

- Incoming packets are classified at the NIC and placed into one of multiple NIC queues; each traffic class is assigned to one or more queues and different traffic classes are mapped to different queues.
- All applications within a hyperapp run on the same core. Hyperapps whose load exceeds a single core's capacity are instantiated on multiple cores (e.g., HyperApp3 in Figure 8). Each core may be assigned one or more hyperapps.
- Each hyperapp instance has a corresponding pshim instance running on the same core and each pshim reads packets from a dedicated virtual NIC queue. For example, HyperApp3 in Figure 8 runs on Core2 and Core3 and has two separate pshims.[7]

The resulting design has several desirable properties conducive to achieving high performance:

- A packet is processed in its entirety on a single core (avoiding inter-core synchronization overheads).

[7]The traffic split between the two instances of HyperApp3 also occurs in the NIC using filters as in §6.1.

- We introduce no shared data structures across cores (avoiding needless cache invalidations).
- There is no contention for access to NIC queues (avoiding the overhead of locking).
- Policy enforcement is lightweight (stateless and requiring no marking or modification of packets).

6 Implementation

Next, we describe how we prototype the different components of the CoMb architecture.

6.1 CoMb Controller

We implement the controller's algorithms using an off-the-shelf solver (CPLEX). The controller runs a pre-processing step to generate the hyperapps and their effective resource footprints taking into account the affinity of applications for specific accelerators. The controller periodically runs the optimization step that takes as inputs the current per-application-port traffic matrix (i.e., per ingress-egress pair), the traffic of interest to each application, the cross-application policy ordering constraints, and the resource footprints per middlebox module.

After running the optimization, it maps the $d_{c,n}$ values to device-level configurations as follows. If the CoMb box supports in-hardware classification (like our prototype server) and has a sufficient number of filter entries, the controller maps the $d_{c,n}$ values into a set of non-overlapping *traffic filters*. As a simple example, suppose c denotes all traffic from sources in 10.1.0.0/16 to destinations in 10.2.0.0/16, and $d_{c,n_1} = d_{c,n_2} = 0.5$. Then the filters for $n_1 : \langle 10.1.0.0/17, 10.2.0.0/16 \rangle$ and $n_2 : \langle 10.1.128.0/17, 10.2.0.0/16 \rangle$.[8] One subtle issue is that it also installs filters corresponding to traffic in the reverse direction. Note that if each CoMb box is off-path, these filters can be pushed to the upstream router or switch.

If the NIC does not support such expressive filters, or has a limited number of filter entries (e.g., if the number of prefix pairs is very high in a large network), the controller falls back to a hash-based configuration [37]. In this case, the basic classification to identify the required hyperapp (say based on the port numbers) still happens at the NIC. The subsequent decision on whether this node is responsible for processing this session happens in the software pshim. Each device's pshim does a fixed-length (/16) prefix lookup, computes a direction-invariant *hash* of the IP 5-tuple [39], and checks if this hash falls in its assigned range. For the above example, the configuration will be $n_1 : \langle 10.1.0.0/16, 10.2.0.0/16, hash \in [0, 0.5] \rangle$ and $n_2 : \langle 10.1.0.0/16, 10.2.0.0/16, hash \in [0.5, 1] \rangle$.

[8]This simple example assumes a uniform distribution of traffic per prefix block. In practice, the prefixes can be weighted by expected traffic volumes inferred from past measurements.

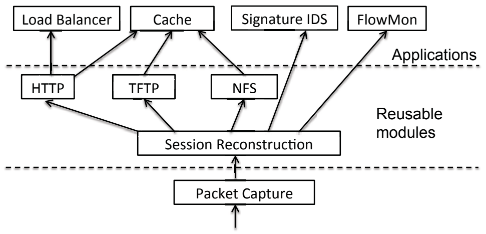


Figure 9: *Our modular middlebox implementation*

6.2 CoMb box prototype

We prototype a CoMb box on a general-purpose server (without accelerators) with two Intel(R) Xeon(R) 'Westmere' CPUs each with four cores at 3.47GHz (X5677) and 48GB memory, configured with four Intel(R) 82599 10 GigE NIC ports [4] each capable of supporting up to 128 queues, running Linux (kernel v.2.6.24.7).

Classification: We leverage the classification capabilities on the NIC. The NIC demultiplexes packets into separate in-hardware queues per hyperapp based on the filters from the controller. The 82599 NIC supports 32K classification entries over: src/dst IP addresses, src/dst ports, protocol, VLAN header, and a flexible 2-byte tuple anywhere in the first 64 bytes of the packet [4]. We use the address and port fields to create filter entries.

Policy Enforcer: We implement the pshim in kernel-mode SMP-Click [30] following the design in §5. In addition to the policy routing, the pshim implements two additional functions: (1) creating interfaces for the application-level processes to receive and send packets (see below) and (2) the optional hash-based check to decide whether to process a specific packet.

6.3 CoMb applications

Our prototype supports two application architectures: modular middlebox applications written in Click and standalone middlebox processes (e.g., Snort, Squid).

Modular middlebox applications: As a proof-of-concept prototype, we implement several canonical middlebox applications: signature-based intrusion detection, flow-level monitoring, a caching proxy, and a load balancer as (user-level) modules in Click as shown in Figure 9. As such, our focus is to demonstrate the feasibility of building modular middlebox applications and establish the potential for reuse. We leave it to future work to explore the choice of an ideal software architecture and an optimal set of reusable modules.

To implement these applications, we port the logic for session reconstruction and protocol parsers (e.g., HTTP and NFS) from Bro [32]. We implement a custom flow monitoring system. Our signature-based IDS uses Bro's signature matching module. We also built a custom Click module for parsing TFTP traffic. The load balancer is a layer-7 application that assigns HTTP requests to different backend servers by rewriting packets. The cache mimics actions in a caching proxy (i.e., storing and looking up requests in cache), but does not rewrite packets.

Application	Dependency chain	Contribution (%)
Flowmon	Session	73
Signature	Session	26
Load Balancer	HTTP,Session	88
Cache	HTTP,Session	54
Cache	NFS,Session	50
Cache	TFTP,Session	36

Table 2: *Contribution of reusable modules*

While Bro's modular design made it a very useful starting point, its intended use is as a standalone IDS while CoMb envisions reusing modules across *multiple applications* from *different vendors*. This leads to one key architectural difference. Modules in Bro are tightly integrated; lower layers are aware of the higher layers using them and "push" data to them. We avoid this tight coupling between the modules and instead implement a "pull" model where lower layers expose well-defined interfaces using which higher-layer functions obtain relevant data structures.

Supporting standalone applications: Last, we focus on how a CoMb box supports standalone middlebox applications (e.g., Snort, Squid). We run standalone applications as separate processes that can access packets in one of two modes. If we have access to the application source, we modify the packet capture routines; e.g., in Snort we replace `libpcap` calls with a memory read to a shared memory region into which the pshim copies packets. For applications where we do not have access to the source, we simply create virtual network interfaces and the pshim writes to these interfaces. The former approach is more efficient but requires source modifications; the latter is less efficient but allows us to run legacy software with no modifications.

7 Evaluation

Our evaluation addresses the following high-level questions regarding the benefits and overheads of CoMb:

- **Single-box benefits:** What reuse benefits can consolidation provide? (§7.1)
- **Single-box overhead:** Does consolidating applications affect performance and extensibility? (§7.2)
- **Network-wide benefits:** What benefits can network administrators realize using CoMb? (§7.3)
- **Network-wide overhead:** How practical and efficient is CoMb's controller? (§7.4)

7.1 Potential for reuse

First, we measure the potential for processing reuse achievable by refactoring middlebox applications. As

§3.2 showed, the savings from reuse depend both on the processing footprints of reusable modules and the expected amount of traffic overlap. Here, we focus only on the former and defer the combined effect to the network-wide evaluation (§7.3). We use real packet traces with full payloads for these benchmarks.[9] Because we are only interested in the *relative contribution*, we run these benchmarks with a single userlevel thread in Click. We use PAPI[10] to measure the number of CPU cycles per-packet each module uses. Note that an application like Cache uses different processing chains (e.g., Cache-HTTP-session vs. Cache-NFS-session); the relative contribution depends on the exact sequence. Table 2 shows that the reusable modules contribute 26–88% of the overall processing across the different applications.

7.2 CoMb single-box performance

We tackle three concerns in this section: (1) What *overhead* does CoMb add for running individual applications? (2) Does CoMb *scale* well as traffic rates increase? and (3) Does application performance suffer when administrators want to *add new functionality*?

For the following experiments, we report throughput measurements using the same full-payload packet traces from §7.1 on our prototype CoMb server with two Intel Westmere CPUs each with four cores at 3.47GHz (X5677) and 48GB memory. The results are consistent with other synthetic traces as well.

7.2.1 Shim Overhead

Recall from §6 that CoMb supports two types of middlebox software: (1) standalone applications (e.g., Snort), and (2) modular applications in Click. Table 3 shows the overhead of running a representative middlebox application from each class on a single core in our platform. We show two scenarios, one where all classification occurs in hardware (labeled *shim-simple*) and when the pshim runs an additional hash-based check as discussed in §6 (labeled *shim-hash*). For middlebox modules in Click, *shim-simple* imposes zero overhead. Interestingly, the throughput for Snort is better than its native performance. The reason is that Click's packet capture routines are more efficient than native Snort (`libpcap` or `daq`). We also see that *shim-hash* adds only a small overhead over *shim-simple*. This result confirms that running applications in CoMb imposes minimal overhead.

7.2.2 Performance under consolidation

Next, we study the effect of adding more cores and adding more applications. For brevity, we only show results for shim-simple. For these experiments, we use

Application architecture	Overhead (%)	
(instance)	Shim-simple	Shim-hash
Standalone (Snort)	-61	-58
Modular (IPSec)	0	0.73
Modular (RE [11])	0	0.62

Table 3: *Performance overhead of the shim layer for different middlebox applications*

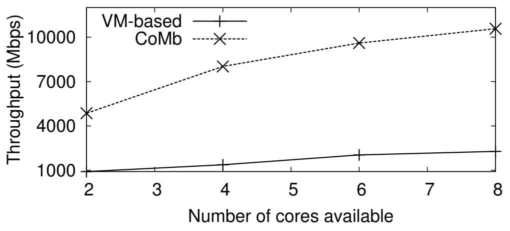


Figure 10: *Throughput vs. number of cores*

a standalone application process using the Snort IDS. To emulate adding new functionality, we create duplicate instances of Snort. (We find similar results with heterogeneous applications too.) At a high-level, we find that consolidation in CoMb does not introduce contention bottlenecks across applications.

As a point of comparison, we also evaluate a *virtual appliance* architecture [22], where each Snort instance runs in a separate VM on top of the Xen hypervisor. To provide high I/O throughput to the VM setup, we use the SR-IOV capability in the hardware [8] and the vSwitching capability of the NIC to transfer packets between application instances [4]. We confirmed that I/O was not a bottleneck; we were able to achieve a throughput of around 7.8 Gbps on a single VM with a single CPU core which is consistent with state-of-art VM I/O benchmarks [40]. As in §5.2, we need to decide between the app-per-core vs. hyperapp-per-core design for the VM setup. We saw that app-per-core is roughly $2\times$ better for the VM case because context switches between VMs are expensive and packet switching between VMs is in hardware (i.e. vSwitching). Thus, we conservatively use the app-per-core design for the VM setup.

Figure 10 shows the effect of adding more cores to the platform with a fixed hyperapp consisting of two Snort processes in sequence. We make three main observations. First CoMb's throughput with this real IDS/IPS is $\approx$ 10 Gbps on a 8-core platform which is comparable to vendor datasheets [2]. Second, CoMb exhibits a reasonable scaling property similar to prior results on multicore platforms [14]. This suggests that adapting CoMb to higher traffic rates simply requires more processing cores and does not need significant re-engineering. Finally, CoMb's throughput is $5\times$ better than the VM case. This performance gap arises out of a combination of three factors. First, Snort atop the pshim runs significantly faster than native Snort because Click offers more effi-

[9]From https://domex.nps.edu/corp/scenarios/2009-m57/net/; we are not aware of other traces with full payloads.

[10]http://icl.cs.utk.edu/papi/

cient packet capture as we saw in Table 3. Second, running Snort inside a VM has roughly 30% lower throughput than native Snort. Third, our hyperapp model amortizes the fixed cost of copying packets into the application layer whereas the VM-based setup incurs multiple copies. While the performance of virtual network appliances is under active research, these results are consistent with benchmarks for virtual appliances [1].

We also evaluated the impact of running more *applications* per-packet. The ideal degradation when we run k applications is a $\frac{1}{k}$ curve because running k applications needs k-times as much work. We found that both CoMb and the VM-based setup have a near-ideal throughput degradation (now shown). This confirms that CoMb allows administrators to easily add new middlebox functionality in response to policy or workload changes.

7.3 Network-wide Benefits

Setup: Next, we evaluate the network-wide benefits that CoMb offers via reuse, multiplexing, and spatial distribution. For this evaluation, we use real topologies from educational backbones and the Enterprise network, and PoP-level AS topologies from Rocketfuel [38]. To obtain realistic time-varying traffic patterns, we use the following approach. We use traffic matrices for Internet2[11] to compute empirical variability distributions for each element in a traffic matrix; e.g., the probability that the volume is between 0.6 and 0.8 the mean. Then, using these empirical distributions, we generate time-varying traffic matrices for the remaining AS-level topologies using a gravity model to capture the mean volume [34]. For the Enterprise network, we replay real traffic matrices.

In the following results, we report the benefits that CoMb provides relative to today's standalone middlebox deployments with the four applications from Table 2: flow monitoring, load balancer, IDS, and cache. To emulate current deployments, we use the same applications but without reusing modules. For each application, we use public configurations to identify the application ports of traffic they process. To capture changes in per-port volume over time, we replay the empirical variability based on flow-level traces from Internet2. We begin with a scenario where all four applications can be spatially distributed and then consider a scenario when two of the applications are topologically constrained.

Provisioning: We consider a *provisioning* exercise to minimize the resources needed to handle the time-varying traffic patterns generated as described earlier across 200 epochs. The metric of interest here is the *relative savings* that CoMb provides vs. today's deployments where all applications run as independent devices only at

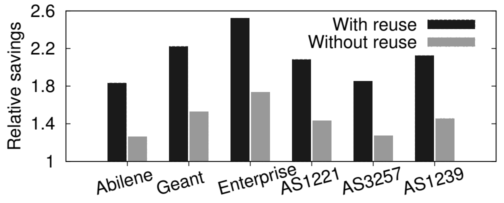


Figure 11: *Reduction in provisioning cost with CoMb*

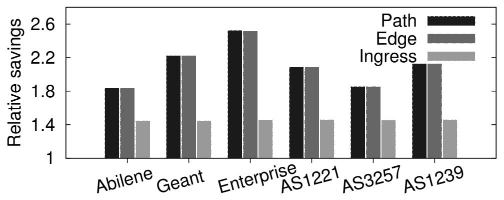


Figure 12: *Impact of spatial distribution on CoMb's reduction in provisioning cost*

the ingress: $\frac{Cost_{standalone,ingress}}{Cost_{CoMb}}$. The *Cost* term here represents the total cost of provisioning the network to handle the given set of time-varying workloads (i.e., $\sum_{n,r} Prov_{n,r}$ from §4).

We try two CoMb configurations: with and without reusable modules. In the latter case, the middlebox applications share the same hardware but not software. Figure 11 shows that across the different topologies CoMb with reuse provides 1.8–2.5$\times$ savings relative to today's deployment strategies. For the Enterprise setting, CoMb even without reuse provides close to 1.8$\times$ savings.

Figure 12 studies the impact of spatial distribution by comparing three strategies for distributing middlebox responsibilities: full path (labeled *Path*), either ingress or egress (labeled *Edge*), or only the *Ingress*. Interestingly, *Edge* is very close to *Path*. To explore this further, we also tried a strategy of picking a *random* second node for each path. We found that this is again very close to *Path* (not shown). In other words, for *Edge* the egress is not special; the key is having *one more* node to distribute the load. We conjecture that this is akin to the "power of two random choices" observation [29] and plan to explore this in future work.

Load balancing: CoMb also allows middlebox deployments to better adapt to changing traffic workloads under a fixed provisioning strategy. Here, our metric of interest is the maximum load across the network, and we measure the *relative* benefit as: $\frac{MaxLoad_{standalone,ingress}}{MaxLoad_{CoMb}}$. We consider two network-wide provisioning strategies where each location is provisioned with the same resources (labeled *Uniform*) or proportional to the average volume it sees (labeled *Volume*). For the standalone case, we assume resources are split between applications proportional to their workload. Note that the combination of volume and workload proportional provisioning likely reflects current practice. We also consider the Uniform case be-

[11] http://www.cs.utexas.edu/~yzhang/research/AbileneTM

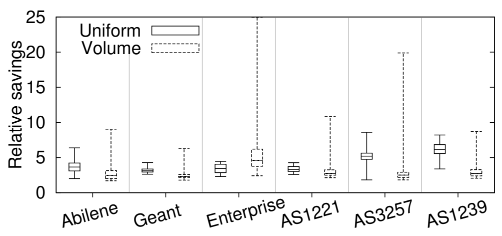


Figure 13: *Relative reduction in the maximum load*

Topology	Unconstrained	Two-step	Ingress-only
Internet2	1.81	1.62	1.41
Geant	2.20	1.71	1.42
Enterprise	2.58	1.76	1.45
AS1221	2.17	1.69	1.41
AS3257	1.85	1.63	1.42
AS1239	2.11	1.69	1.43

Table 4: *Relative savings in provisioning when Cache and Load balancer are spatially constrained*

cause it is unclear if the proportional allocation strategy is always better for today's deployments; e.g., it could be better on average, but have worse "tail" performance as shown in Figure 13.

As before, we generate time-varying traffic patterns over 200 epochs and measure the above relative load metric per epoch. For each topology, Figure 13 shows the distribution of this metric (across epochs) using a box-and-whiskers plot with the 25^{th}, 50^{th}, and 75^{th} percentiles, and the minimum and maximum values. The result shows that CoMb reduces the maximum load by $> 2\times$ and the reduction can be as high as $25\times$, confirming that CoMb can better handle traffic variability compared to current middlebox deployments.

Topological constraints: Next, we consider a scenario when some applications cannot be spatially distributed. Here, we constrain Cache and the Load balancer to run at the ingress for each path. One option in this scenario is to pin all middlebox applications to the ingress (to exploit reuse) but ignore spatial distribution. While CoMb provides non-trivial savings ($1.4\times$) even in this case, this ignores opportunities for further savings. To this end, we extend the formulation from §4.2 to perform a two-step optimization. In the first step, we assign the topologically constrained applications to their required locations. In the second, we assign the remaining applications, which can be distributed, with a slight twist. Specifically, we reduce the hyperapp-footprints on locations where they can reuse modules with the constrained applications. For example, if we have the hyperapp Cache-IDS, with Cache pinned to the ingress, we reduce the IDS footprint on the ingress. Table 4 shows that this two-step procedure is able to improve the savings 20–30% compared to an ingress-only solution. This preliminary analysis suggests that CoMb can work even when some applications are topologically constrained. As future work, we plan to explore a detailed analysis of such topological constraints.

Topology	Path	Edge	Ingress
Internet2	0.87	0.87	0.54
Geant	1.49	1.25	0.55
Enterprise	1.02	1.02	0.54
AS1221	1.33	1.33	0.54
AS3257	0.68	0.68	0.55
AS1239	1.26	1.26	0.55

Table 5: *Relative size of the largest CoMb box. A higher value here means that the standalone case needs a larger box compare to CoMb*

Topology	#PoPs	Time (s)	
		Strawman-LP	hyperapp
Internet2	11	687.68	0.05
Geant	22	3455.28	0.24
Enterprise	23	2371.87	0.25
AS3257	41	1873.32	0.78
AS1221	44	3145.77	1.08
AS1239	52	9207.78	1.58

Table 6: *Time to compute the optimal solution*

Does CoMb need bigger boxes? A final concern is that consolidation may require "beefier" boxes; e.g., in the network core. To address this concern, Table 5 compares the processing capacity of the largest standalone box needed across the network to that of the largest CoMb box: $\frac{Largest_{standalone}}{Largest_{CoMb}}$. We see that the largest standalone box is actually *larger* than CoMb for many topologies. Even without distribution, the largest CoMb box is only $\frac{1}{0.55} = 1.8\times$, which is quite manageable.

7.4 CoMb controller performance

Last, we focus on two key concerns surrounding the performance of the CoMb network controller: (1) Is the optimization fast enough to respond to traffic dynamics (on the order of a few minutes)? and (2) How close to the theoretical optimal is our formulation from §4.2?

Table 6 shows the time to run the optimization from Section 4 using the CPLEX LP solver on a single core Intel(R) Xeon(TM) 3.2GHz CPU. To put our formulation in context, we also show the time to solve an LP relaxation for a precise model that captures reuse and policy dependencies on a per-session and per-action basis [35]. (The precise model is an intractable discrete optimization problem; we use its LP relaxation as a proxy.) The result shows that our formulation is almost four orders of magnitude faster than the precise model. Given that we expect to recompute configurations on the order of a few minutes [20], the compute times (1.58s for a 52-node topology) are reasonable.

We also measured the optimality gap between the precise formulation [35] and our practical approach over a range of scenarios. Across all topologies, the optimality gap is $\leq 0.19\%$ for the load balancing and $\leq 0.1\%$ for the provisioning (not shown). Thus, our formulation provides a tractable, yet near-optimal, alternative.

8 Discussion

The consolidated middlebox architecture we envision raises two deployment concerns that we discuss next.

First, CoMb changes existing business and operating models for middlebox vendors as it envisions vendors that decouple middlebox software and hardware and also those who refactor their applications to exploit reuse. We believe that the qualitative (i.e., extensibility, reduced sprawl, and simplified management) and quantitative (i.e., lower provisioning costs and better resource management) advantages that CoMb offers will motivate vendors to consider product offerings in this space. Furthermore, evidence suggests vendors are already rethinking the software-hardware coupling and starting to offer software-only "virtual appliances" (e.g., [7]). We also speculate that some middlebox vendors may already internally have modular middlebox stacks. CoMb simply requires one or more of these vendors to provide open APIs to these modules to encourage further innovation.

Second, running multiple middlebox applications on the same platform raises concerns about *isolation* with respect to performance (e.g., contention for resources), security (e.g., the NIDS/firewall must not be compromised), and fault tolerance (e.g., a faulty application should not crash the whole system). With respect to performance, concurrent work shows that contention has minimal impact on throughput on x86 hardware for the types of network processing workloads we envision [16]. In terms of fault tolerance and security, process boundaries already provide some degree of isolation and techniques such as containers can give stronger properties [5]. There are two challenges with such sandboxing. The first is ensuring the context switching overheads are low. Second, even though CoMb without reusing modules provides significant benefits, it would be useful to provide isolation without sacrificing the benefits of reuse. We also note that running applications in user space can further insulate misbehaving applications. In this light, recent results showing the feasibility of high performance network I/O in the user space are promising [33].

9 Related Work

Integrating middleboxes: Previous work discusses mechanisms to better expose middleboxes to administrators (e.g., [6]). Similarly, Joseph et al. describe a switching layer for integrating middleboxes in datacenters [26]. CoMb focuses on the orthogonal problem of consolidating middlebox deployments.

Middlebox measurements: Studies have measured the end-to-end impact of middleboxes [10] and interactions with transport protocols [24]. The measurements in §2 and high-level opportunities in §3 appeared in an earlier workshop paper [36]. This work goes beyond the motivation to demonstrate a practical design and implementation and quantifies the single-box and network-wide benefits of a consolidated middlebox architecture.

General-purpose network elements: There are many efforts to build commodity routers and switches using x86 CPUs [18, 19, 22], GPUs [23], and merchant switch silicon [27]. CoMb can benefit from these advances as well. It is worth noting that the challenges we address in CoMb also apply to these efforts, if the extensibility they enable leads to diversity in traffic processing.

Rethinking middlebox design: CoMb shares the motivation of rethinking middlebox design with Flowstream [22] and xOMB [12]. FlowStream presents a high-level architecture using OpenFlow for policy routing and runs each middlebox as a VM [22]. Unlike CoMb, a VM approach precludes opportunities for reuse. Further, as §7.2 shows today's VM-based solutions have considerably lower throughput. xOMB presents a software model for extensible middleboxes [12]. CoMb addresses network-wide and platform-level resource management challenges that arise with consolidation that neither FlowStream nor xOMB seek to address. CoMb also provides a more general management framework to support both modular and standalone middlebox functions.

Network management: CoMb's controller follows in the spirit of efforts showing the benefits of centralization in routing, access control, and monitoring (e.g., [21, 31, 15]). The use of optimization arises in other network management applications. However, reuse and policy dependencies that arise in the context of consolidating middlebox management create new challenges for management and optimization unique to our context.

10 Conclusions

We presented a new middlebox architecture called CoMb, which systematically explores opportunities for *consolidation*, both in building individual appliances and in managing an ensemble of these across a network. In addition to the qualitative benefits with respect to extensibility, ease of management, and reduction in device sprawl, consolidation provides new opportunities for resource savings via application multiplexing, software reuse, and spatial distribution. We addressed key resource management and implementation challenges in order to leverage these benefits in practice. Using a prototype implementation in Click, we show that CoMb reduces the network provisioning cost by up to $2.5\times$, decreases the load skew by up to 25×, and imposes minimal overhead for running middlebox applications.

Acknowledgments

We thank Katerina Argyraki, Sangjin Han, and our shepherd Miguel Castro for their feedback on the paper. We also thank Mihai Dobrescu and Vitaly Luban for helping us with experiment setup. This work was supported in part by ONR grant N000141010155, NSF grants 0831245 and 1040626, and the Intel Science and Technology Center for Secure Computing.

11 References

[1] IBM Network Intrusion Prevention System Virtual Appliance. `http://www-01.ibm.com/software/tivoli/products/virtualized-network-security/requirements.html`.

[2] Big-IP hardware datasheet. `www.f5.com/pdf/products/big-ip-platforms-ds.pdf`.

[3] Crossbeam network consolidation. `http://www.crossbeam.com/why-crossbeam/network-consolidation/`.

[4] Intel 82599 10 gigabit ethernet. `http://download.intel.com/design/network/datashts/82599_datasheet.pdf`.

[5] Linux containers. `http://lxc.sourceforge.net/man/lxc.html`.

[6] Middlebox Communications (MIDCOM) Protocol Semantics. RFC 3989.

[7] Silver Peak releases software-only WAN optimization . `http://www.networkworld.com/news/2011/071311-silver-peak-releases-software-only-wan.html`.

[8] SR-IOV. `http://www.pcisig.com/specifications/iov/single_root/`.

[9] World Enterprise Network and Data Security Markets. `http://www.abiresearch.com/press/3591-Enterprise+Network+and+Data+Security+Spending+Shows+Remarkable+Resilience`.

[10] M. Allman. On the Performance of Middleboxes. In *Proc. IMC*, 2003.

[11] A. Anand, A. Gupta, A. Akella, S. Seshan, and S. Shenker. Packet Caches on Routers: The Implications of Universal Redundant Traffic Elimination. In *Proc. of SIGCOMM*, 2008.

[12] J. Anderson. *Consistent Cloud Computing Storage as the Basis for Distributed Applications*, chapter 7. University of California, San Diego, 2012.

[13] T. Benson, A. Akella, and A. Shaikh. Demystifying configuration challenges and trade-offs in network-based isp services. In *Proc. SIGCOMM*, 2011.

[14] S. Boyd-Wickizer, A. T. Clements, Y. Mao, A. Pesterev, M. F. Kaashoek, R. Morris, and N. Zeldovich. An Analysis of Linux Scalability to Many Cores. In *Proc. OSDI*, 2010.

[15] M. Caesar, D. Caldwell, N. Feamster, J. Rexford, A. Shaikh, and J. van der Merwe. Design and implementation of a Routing Control Platform. In *Proc. of NSDI*, 2005.

[16] M. Dobrescu, K. Argyarki, and S. Ratnasamy. Toward Predictable Performance in Software Packet-Processing Platforms. In *Proc. NSDI*, 2012.

[17] M. Dobrescu, K. Argyarki, M. Manesh, G. Iannaccone, and S. Ratnasamy. Controlling Parallelism in Multi-core Software Routers. In *Proc. PRESTO*, 2010.

[18] M. Dobrescu, N. Egi, K. Argyraki, B.-G. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy. RouteBricks: Exploiting Parallelism to Scale Software Routers. In *Proc. SOSP*, 2009.

[19] N. Egi, A. Greenhalgh, M. Handley, M. Hoerdt, F. Huici, and L. Mathy. Towards high performance virtual routers on commodity hardware. In *Proc. CoNEXT*, 2008.

[20] A. Feldmann, A. G. Greenberg, C. Lund, N. Reingold, J. Rexford, and F. True. Deriving Traffic Demands for Operational IP Networks: Methodology and Experience. In *Proc. of ACM SIGCOMM*, 2000.

[21] A. Greenberg, G. Hjalmtysson, D. A. Maltz, A. Meyers, J. Rexford, G. Xie, H. Yan, J. Zhan, and H. Zhang. A Clean Slate 4D Approach to Network Control and Management. *ACM SIGCOMM CCR*, 35(5), Oct. 2005.

[22] A. Greenlagh, M. Handley, M. Hoerdt, F. Huici, L. Mathy, and P. Papadimitriou. Flow Processing and the Rise of Commodity Network Hardware. *ACM CCR*, Apr. 2009.

[23] S. Han, K. Jang, K. Park, and S. Moon. PacketShader: a GPU-Accelerated Software Router. In *Proc. SIGCOMM*, 2010.

[24] M. Honda, Y. Nishida, C. Raiciu, A. Greenhalgh, M. Handley, and H. Tokuda. Is it still possible to extend TCP? In *Proc. IMC*, 2011.

[25] D. Joseph and I. Stoica. Modeling middleboxes. *IEEE Network*, 2008.

[26] D. A. Joseph, A. Tavakoli, and I. Stoica. A Policy-aware Switching Layer for Data Centers. In *Proc. SIGCOMM*, 2008.

[27] G. Lu, C. Guo, Y. Li, Z. Zhou, T. Yuan, H. Wu, Y. Xiong, R. Gao, and Y. Zhang. ServerSwitch: A Programmable and High Performance Platform for Data Center Networks. In *Proc. NSDI*, 2011.

[28] Y. Ma, S. Banerjee, S. Lu, and C. Estan. Leveraging Parallelism for Multi-dimensional Packet Classification on Software Routers. In *Proc. SIGMETRICS*, 2010.

[29] M. Mitzenmacher, A. W. Richa, and R. Sitaraman. The Power of Two Random Choices: A Survey of Techniques and Results . Handbook of Randomized Computing, 2000.

[30] R. Morris, E. Kohler, J. Jannotti, and M. F. Kaashoek. The click modular router. *SIGOPS Operating Systems Review*, 33(5):217–231, 1999.

[31] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. OpenFlow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, 2008.

[32] V. Paxson. Bro: A System for Detecting Network Intruders in Real-Time. In *Proc. USENIX Security Symposium*, 1998.

[33] L. Rizzo, M. Carbone, and G. Catalli. Transparent acceleration of software packet forwarding using netmap. In *Proc. Infocom*, 2012.

[34] M. Roughan. Simplifying the Synthesis of Internet Traffic Matrices. *ACM SIGCOMM CCR*, 35(5), 2005.

[35] V. Sekar, N. Egi, S. Ratnasamy, M. K. Reiter, and G. Shi. Design and implementation of a consolidated middlebox architecture. Technical Report UCB/EECS-2011-110, EECS Department, University of California, Berkeley, Oct 2011.

[36] V. Sekar, S. Ratnasamy, M. K. Reiter, N. Egi, and G. Shi. The Middlebox Manifesto: Enabling Innovation in Middlebox Deployments. In *Proc. HotNets*, 2011.

[37] V. Sekar, M. K. Reiter, W. Willinger, H. Zhang, R. Kompella, and D. G. Andersen. cSamp: A System for Network-Wide Flow Monitoring. In *Proc. of NSDI*, 2008.

[38] N. Spring, R. Mahajan, and D. Wetherall. Measuring ISP Topologies with Rocketfuel. In *Proc. of ACM SIGCOMM*, 2002.

[39] M. Vallentin, R. Sommer, J. Lee, C. Leres, V. Paxson, and B. Tierney. The NIDS Cluster: Scalable, Stateful Network Intrusion Detection on Commodity Hardware. In *Proc. of RAID*, 2007.

[40] H. Zhiteng. I/O Virtualization Performance. `http://xen.org/files/xensummit_intel09/xensummit2009_IOVirtPerf.pdf`.

An Operating System for the Home

Colin Dixon *(IBM Research)* Ratul Mahajan Sharad Agarwal
A.J. Brush Bongshin Lee Stefan Saroiu Paramvir Bahl

Microsoft Research

Abstract—Network devices for the home such as remotely controllable locks, lights, thermostats, cameras, and motion sensors are now readily available and inexpensive. In theory, this enables scenarios like remotely monitoring cameras from a smartphone or customizing climate control based on occupancy patterns. However, in practice today, such smarthome scenarios are limited to expert hobbyists and the rich because of the high overhead of managing and extending current technology.

We present HomeOS, a platform that bridges this gap by presenting users and developers with a PC-like abstraction for technology in the home. It presents network devices as peripherals with abstract interfaces, enables cross-device tasks via applications written against these interfaces, and gives users a management interface designed for the home environment. HomeOS already has tens of applications and supports a wide range of devices. It has been running in 12 real homes for 4–8 months, and 42 students have built new applications and added support for additional devices independent of our efforts.

1 Introduction

Pop culture, research prototypes and corporate demos have all envisioned a smart, connected home where multiple devices cooperate to cater to users' wishes with little or no effort. For instance, in a home with remotely controllable lights, cameras and locks, it should be easy to automatically adjust lights based on the weather and time of day as well as remotely view who is at the door before unlocking it. But such seamless home-wide tasks are conspicuously absent from the mainstream despite the fact that the needed hardware devices are reasonably priced—wireless lightswitches, door locks, and cameras can each be bought for (US) $50–100.

Studies of technology use in the home help explain the gap between the longstanding vision of connected homes and its reality [7, 8, 15, 22, 34]. They find that it is increasingly difficult for users to manage the growing number of devices in their homes. Further, application software that can compose the functionality of these devices is hard to develop because of extreme heterogeneity across homes, in terms of devices, interconnectivity, and user preferences. Finally, finding hardware and software that is compatible with existing home technology is error prone at best. This is problematic as users prefer to organically add a few devices or applications at a time.

We argue that this state of affairs stems directly from the abstractions that home technology presents to users and developers. There are two prevalent abstractions today: an appliance and a network-of-devices. The appliance abstraction is that of a closed, monolithic system supporting a fixed set of tasks over a fixed set of devices. Commercial security and automation systems [1, 12] and many research efforts [29, 43] present this abstraction. The closed nature of such systems means that end users and third-party developers typically cannot extend them, making it a poor fit for an environment where incremental extensions are desired.

The second abstraction is a decentralized network-of-devices. Interoperability protocols such as DLNA [14], Z-Wave [46] and SpeakEasy [16] provide this abstraction. It is also a poor fit for the home because it provides limited or no support for users to manage their technology or for developers to build portable applications that span multiple devices.

In this paper, we advocate for a PC-like abstraction for technology in the home—all devices in the home appear as peripherals connected to a single logical PC. Users and applications can find, access and manage these devices via a centralized operating system. The operating system also simplifies the development of applications by abstracting differences across devices and homes. Further, it provides a central location to extend the home by adding new devices and applications.

We present an architecture to provide the PC abstraction for home technology and its instantiation in the form of a system called HomeOS. Its design is based on user interviews and feedback from a community of real users and developers. It has been under development for over two years.

HomeOS uses (i) Datalog-based access control and other primitives that simplify the task of managing technology in the home, (ii) protocol-independent services to provide developers with simple abstractions to access devices and (iii) a kernel that is agnostic to the devices to which it provides access, allowing easy incorporation of new devices and applications. HomeOS runs on a dedicated computer in the home (e.g., the gateway) and does not require any modifications to commodity devices.

Our current prototype supports several device protocols (e.g., Z-Wave and DLNA) and many kinds of devices (e.g., lights, media renderers and door/window sensors). It runs in 12 real homes and 42 students have developed applications using it. These homes run applications varying from getting e-mail notifications with photos when the front or back door is opened at unexpected times, to seamlessly migrating video around the house. Students have built applications ranging from using Kinect cameras to control devices via gestures to personalized, face-recognition-based reminder systems.

The experiences of these users and developers, along with our controlled experiments, help validate the usefulness of the PC abstraction and our design. Users were able to easily manage HomeOS and particularly liked the ability to organically add devices and applications to their deployments. Developers appreciated the ease with which they could implement desired functionality in HomeOS, without worrying about low-level details of devices and protocols. These experiences also point toward avenues for future work where we could not provide a clean PC abstraction. For instance, connectivity to network devices, especially wireless ones, is harder to diagnose than for directly connected PC peripherals.

In summary, we make three main contributions. First, we propose using a PC abstraction for technology in the home to improve manageability and extensibility. Second, we implement this abstraction in HomeOS. While we do not claim the pieces of our design are novel, to our knowledge, their use in addressing the challenges of the home environment is novel. Third, we validate the PC abstraction and our design with both controlled experiments and real users and developers.

2 A new abstraction for home technology

Our proposal to use a PC-like abstraction for technology in the home is motivated by our own recent study of home technology [7] as well as the work of others [8, 15, 22, 34]. We first summarize the challenges uncovered by this work, then explain why existing abstractions for home technology cannot meet those challenges, and finally present the PC abstraction.

2.1 Challenges

Home technology faces three main challenges today.

1. Management Unlike other contexts (e.g., enterprise or ISP networks), the intended administrators are non-expert users. But the management primitives available to users today were originally designed for experts. As a result, most users find them hard to use. Worse, devices often need to be individually managed and each comes with its own interface and semantics, rather than having a single, unified interface for the home.

The management challenge is particularly noteworthy when it comes to security and access control where users are frequently forced to choose between inconvenience and insecurity [7, 22]. When they are unable to easily and securely configure guest access for devices (e.g., printers) on their home networks, they either deny access to guests or completely open up their networks.

2. Application development Users want to compose their devices in various ways [34] and software should be able to do just that, but heterogeneity across homes makes it difficult to develop such application software. We identify four primary sources of heterogeneity.

- *Topology:* Devices are interconnected in different ways across homes. Some homes have a Wi-Fi-only network while others have a mix of Wi-Fi, Ethernet and Z-Wave. Further, some devices use multiple connectivity modes (e.g., smartphones switch between home Wi-Fi and 3G).
- *Devices:* Different devices, even of the same type, support different standards. For example, light switches may use Z-Wave, ZigBee or X10; and TVs use DLNA, UPnP A/V or custom protocols.
- *User control:* Different homes have different requirements as to how activities should occur [22]. Some homes want the Xbox off after 9 PM and some want security cameras to record only at night.
- *Coordination:* If multiple tasks are running, simultaneous accesses to devices will inevitably arise. Such accesses may be undesirable. For instance, a climate control application may want the window open when a security application wants it closed.

3. Incremental growth Users frequently want to grow their technology incrementally, as their preferences evolve [7, 22]. Such growth is difficult today because users cannot tell if a given piece of technology will be compatible with what they currently have. This difficulty corners them into buying from one vendor (creating lock-in), seeking expensive professional help, and making significant upfront investments (e.g., buying a home-wide automation system with many features before knowing which features fit their lifestyle). Supporting incremental growth is further complicated by the rapid innovation in hardware and software; users' existing systems frequently do not support these new technologies.

2.2 Prevalent abstractions

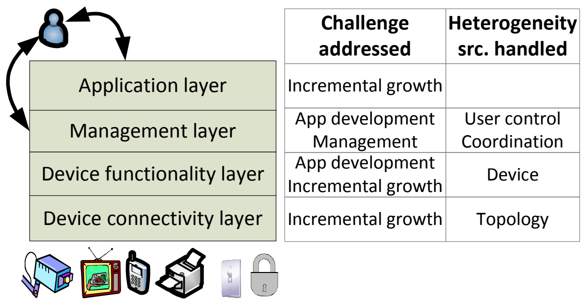
Today, home technology can be seen as presenting one of two abstractions to users and developers. The first is the appliance abstraction that provides the same interface that a monolithic, fixed-function device would. It is used for most home security and home automation systems where the set of devices and tasks are both closed. This has the advantage of offering (potentially) simpler user interfaces and simpler integration across the set of involved devices. However, it inhibits extensibility and application development because integration with third-party devices and software is typically not possible. As a result, the security, audio-video, and automation systems are mutually isolated in many homes [7, 22].

The second abstraction is a network-of-devices, which arises from interoperability protocols offering standardized interfaces to devices. This means that, in theory, applications can remotely control devices and devices can be integrated to accomplish tasks. For instance, DLNA allows some TVs to play media content from a computer. In practice, it leaves too much for users and developers to do for themselves. Users interact with each device's own management interface and application developers must deal with all the sources of heterogeneity mentioned above.

2.3 The PC abstraction

The abstractions prevalent today demonstrate the inherent tension between ease of management on one side and extensibility (for both applications and devices) on the other. The appliance abstraction can provide simple user management (at least for the included devices), but typically does not accommodate new devices and applications. On the other hand, the network-of-devices abstraction readily incorporates new devices, but does not provide the needed support for developing cross-device applications or simple management tools.

We propose to resolve this tension by presenting the abstraction of a PC. Network devices appear as connected peripherals, and tasks over these devices are akin to applications that run on a PC. Users extend their home technology by adding new devices or installing new applications without any guesswork with respect to compatibility. They implement desired access control policies by interacting with the operating system, rather than with individual devices and applications. Finally, applications are written against higher-level APIs, akin to abstract PC driver interfaces, where developers do not have to worry about low-level details of heterogeneous devices and their connectivity. Our proposal is inspired by current PC OSes that make some network devices (e.g., printers) appear local. We take this design to its logical extreme, making all network devices appear local, while tackling other challenges of the home environment.



	Challenge addressed	Heterogeneity src. handled
Application layer	Incremental growth	
Management layer	App development Management	User control Coordination
Device functionality layer	App development Incremental growth	Device
Device connectivity layer	Incremental growth	Topology

Figure 1: Layers in HomeOS and their considerations

3 HomeOS architecture

HomeOS uses a layered architecture (Figure 1) to bridle the complexity of the home environment and address the challenges mentioned in the previous section. Below, we describe each of the four layers in detail, but briefly, the key elements of our approach are: (i) providing users with management primitives and interfaces that align with how they want to manage and secure their home technology, (ii) providing application developers with high-level APIs that are independent of low-level details of devices, and (iii) having a kernel that is independent of specific devices and their functionality. Our design borrows heavily from traditional OSes but also differs from them in a few key ways.

3.1 Device connectivity layer

The Device Connectivity Layer (DCL) solves the problems of discovering and associating with devices. This includes dealing with issues arising from protocols designed to operate only on one subnet (e.g., UPnP) as well as connecting to devices with multiple connectivity paradigms (e.g., a smartphone on WiFi vs. 3G).

The DCL provides higher layers with handles for exchanging messages with devices, but it attempts to be as thin as possible, avoiding any understanding of device semantics. There is one software module in the DCL for each protocol (e.g., DLNA and Z-Wave). This module is also responsible for device discovery, using protocol-specific methods (e.g., UPnP probes). If it finds an unknown device it passes that up to the management layer where the proper action can be taken.

The DCL frees developers from worrying about some of the most pernicious issues in using distributed hard-

Pan, Tilt and Zoom Camera
GetImage() → *bitmap*
GetVideo()† → *bitmaps*
Up()
Down()
Left()
Right()
ZoomIn()
ZoomOut()

DLNA Media Renderer
Play(*uri*)
PlayAt(*uri*, *time*)
Stop()
Status() → *uri*, *time*

Dimmer Switch
Get()† → *percent*
Set(*percent*)

Figure 2: Example HomeOS roles and their operations. '$\dagger$' indicates that the operation can be subscribed to

ware by having a different layer take care of discovering and maintaining connectivity to devices.

3.2 Device functionality layer

The Device Functionality Layer (DFL) takes the handles provided by the DCL and turns them into APIs that developers can easily use. These APIs are services that are independent of device interoperability protocols (§3.2.1), and the DFL is architected to allow easy incorporation of new devices and interfaces whether they are similar to existing ones or not (§3.2.2).

3.2.1 Protocol-independent services

DFL modules provide device functionality to applications using a service abstraction. We refer to service interfaces as *roles*, and each role contains a list operations that can be invoked. For instance, the "lightswitch" role has two operations, "turnOn" and "turnOff," each taking no arguments. Role names are unique with semantics; "lightswitch" implies functionality that is the same across device vendors and homes. Operations may return results immediately and/or allow subscription to events of interest (e.g., when a light switch is physically toggled). Figure 2 shows a few example roles in HomeOS.

In HomeOS, DFL specifications only capture device functionality (and no other detail), and thus the applications that use them do not require changes unless device functionality itself evolves. (We describe below how we handle changing device functionality.) In contrast, the common method today for applications to use network devices is to use a device protocol. For instance, an application might use DLNA to play videos on a remote TV. Using device protocols is problematic because there are many such protocols and they continue to evolve.

We believe that continuous evolution of device protocols is inevitable because they tend to be fat, spanning many layers and concerns. For instance, Z-Wave specifies not only device functionality but also MAC and PHY layer details, and UPnP requires the use of HTTP, SOAP, XML and TCP/IP. As technology changes or new concerns arise (e.g., low-energy, low-bandwidth), new protocols emerge to meet engineering needs.

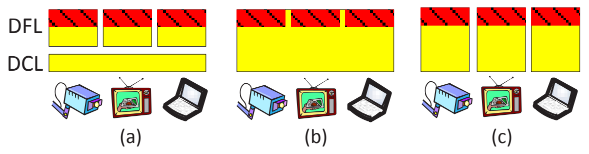


Figure 3: The organization of software modules in HomeOS (a) and two alternative organizations (b) and (c); Solid regions represent protocol-specific code and hatched regions represent protocol-independent code

3.2.2 Extensibility

Introducing new devices in HomeOS is straightforward. A new device can either use an existing role or, if it is a new type of device, a new role can be registered for its functionality. Applications can then be written against this new role, without the need to upgrade HomeOS itself because the kernel is completely agnostic to the services spoken across it. This behavior is different from peripheral APIs in current PC OSes which typically require OS updates to provide common support for new peripherals to application developers. Simplifying the introduction of new functionality is important in the home where new devices arrive frequently. Just in the last five years, devices like depth cameras (Kinect), Internet connected DVRs, and digital photo frames have gone from nearly non-existent to commonplace.

Similarly, if a new capability for an existing device is developed, a new role that exposes the capability can be registered. The DFL module for this device can export both the old and new roles for backward compatibility. Currently, each role in HomeOS is independent; we are considering arranging them in a class hierarchy in which subclasses correspond to more specialized functionality.

3.2.3 Alternative architectures

The HomeOS architecture splits device interactions into two separate layers—a device-agnostic layer (DCL) for basic connectivity and device-specific layer (DFL) that builds on this basic connectivity to mediate between applications and devices. Figure 3(a) illustrates the software organization for three devices that share the same underlying protocol. It is similar to what occurs in PC OSes for USB devices, with a USB-specific module and a device-specific module. However, for network devices, where no universal device protocol exists, this organization may appear ill-advised. The knowledge of a device protocol is spread across two layers, and incorporating a new protocol requires changes to both.

In the home setting, this design has important advantages over alternatives. Figure 3(b) shows one alternative that follows a "one driver per protocol" architecture. It is convenient if there are one or a small number of supported devices in the protocol, but quickly becomes cumbersome to maintain as the number of devices in the protocol grows. A more traditional "one driver per device" architecture produces the software organization in Figure 3(c) where both connectivity and functionality are mediated by a single piece of software for each device. This not only results in DCL functionality being replicated if multiple devices use the same protocol, but also necessitates coordination among modules communicating to different devices. For instance, Z-Wave only allows one live message in the Z-Wave network at a time requiring coordination among any software modules that might send Z-Wave messages. Thus, after experimenting with these alternatives, we chose to split device communication across DCL and DFL as shown in Figure 3(a).

3.3 Management layer

The management layer in HomeOS provides two key functionalities. First, it provides a central place to add and remove applications, devices, and users as well as to specify access control policies. Second, it mediates potentially conflicting accesses to devices ensuring that applications do not need to build their own mechanisms to handle shared devices.

We provide both functionalities using the same primitives. A key goal for their design is that they be simple and translate into management interfaces with which users can easily implement desired policies. Otherwise, we risk not only user frustration but also misconfigurations with serious security and privacy consequences, given the sensitive nature of many devices (e.g., cameras and locks). To gain insights on the design requirements, we conduct a study of households with existing automation. We describe the study in §3.3.1 and the primitives in §3.3.2. While these primitives do not extend the state of the art beyond what researchers have proposed in the past, their simplicity and universal application across an OS goes beyond commodity OSes.

3.3.1 Understanding management requirements

Users have complex needs of home technology. While different than those of experts, their own mental models are often refined. To understand these mental models and common user activities, we visited households with home automation already installed (e.g., remote lighting and locks, security cameras). While we expect HomeOS to enable tasks not possible today, these households can give us insight into their experience with current technology as well as how they would like to manage home technology in the future. We present results on the former elsewhere [7] and focus here on the latter aspect. We interviewed 31 people across 14 homes (1 in the UK, rest in the USA), with different systems such as Elk M1, Control4 and Leviton.

Our visits revealed that households want access control primitives that differ from those present in traditional OSes. We summarize four important differences below.

1. Time-based access control Our participants wanted to control access to devices based on time. Parents mentioned restricting children from using certain devices after certain times (e.g., "If my daughter wanted watch [Curious] George at 11 o'clock at night, I wouldn't want to do that"). While social interaction suffices to address some of these concerns, many parents asked for technical means as well. Time-based access control is also needed to give households an ability to grant a variety of access durations for guests (e.g., a few hours to babysitters and a few days to house guests).

Current commodity OSes provide coarse-grained parental controls that can limit whole accounts to certain times of the day, but they lack flexible controls that can easily implement policies such as those above.

2. Applications as security principals Users highlighted a desire to be able to limit applications' abilities to access devices. One participant said "I don't want to grant it [the application] access to everything, just my laptop." A participant in a different home commented about another application: "if it said my DVR and my TV I would say fine, ... if it had my phone or my computer I would want to be able to choose [what it can access]."

This observation requires treating applications as first-order security principals, in addition to users. In current PC OSes, users alone are the primary security principals (with some exceptions such as firewall rules in Windows), and applications simply inherit users' privileges. While smartphone OSes treat applications as security principals, they are solving the simpler problem of regulating single-user, self-contained resources.

3. Easy-to-understand, queryable settings As expected, users complained about complicated interfaces to configure devices (and especially security), but they also bemoaned the lack of a simple way to verify security settings. They had no way convince themselves that they had correctly configured their settings. For example, to ask if guests can access security devices or if a given application cannot unlock the door after 10 PM.

Providing reliable answers to such questions is difficult in current OSes due to issues such as dynamic delegation [10]. In the home, the consequences of incorrect configurations can be severe, requiring even more confidence in security. The lack of such a capability can scare users away from the idea of using new or potentially dangerous capabilities, even if it is possible they are correctly configured. For instance, a participant with electronic door locks said he had not hooked up remote access because he was not "100% certain of its security."

4. Extra sensitive devices Our users showed heightened sensitivity for the security and use of certain devices (e.g., locks and cameras). They wanted support to ensure accidentally granting access to such devices was difficult.

3.3.2 Primitives

The requirements for security and access control outlined above are in conflict. The first two call for primitives that are richer than those in current OSes. However, non-experts find it hard to configure and understand even those primitives [10, 33]. We reconcile the conflict by noting that the home is a much simpler environment that does not need much of the complexity motivated by enterprise environments (e.g., dynamic delegation, highly-customizable ACLs and exceptions).

Datalog access control rules We formulate access control policies as Datalog [9] rules of the form (r, g, m, T_s, T_e, d, pri, a), which states that resource r can be accessed by users in group g, using module m, in the time window from T_s to T_e, on day of the week d, with priority pri and access mode a. Time window and day of the week lets users specify policies by which something is allowed, for instance, on Sundays 7–9 PM. Groups such as "kids" and "adults" are configured separately. Priorities are used to resolve conflicting access to the same resource. Access mode is one of "allow" or "ask." With the latter, the users have the option to permit or deny access interactively when the access is attempted. Studies show that users prefer this flexibility rather than having to specify all possible legal accesses a priori [5, 33]. Any access that is not in the rule database is denied. While these rules may seem complex for users at first, they are amenable to visualization and English sentences like "Allow residents to access the living room speakers using the music player from 8 AM to 10 PM."

Expressing access control as Datalog rules meets our requirements. Users can configure time-based policies as well as restrict an application to accessing only certain devices. They can also easily understand their configuration by queries such as "Which applications can access the door?", "Which devices can be accessed after 10 PM?" or "Can a user ever access the back door lock?" to fully understand their risk. Reliably providing such views is straightforward because they can be formulated as Datalog queries. Answering these queries is fast despite there being many dimensions per rule. Because the policies are straightforward, as we show later, even non-experts can configure and understand them.

The main advantage of Datalog over ACLs is its simplicity. ACLs can be more expressive, assuming we extend them to include time and applications. But they are hard for users to program [33] and hard to aggregate and summarize. We are not the first to propose the use of Datalog for access control, but its use can require major extensions to accommodate policies of complex environments [32]. We find that the needs of the home environment can be met without such extensions.

Past systems looking to simplify access control have explored using a simple table [38] with the principals along one axis and the objects along the other with each cell specifying if access should be allowed. While promising, this approach does not scale well beyond two dimensions and our interviews indicated that time and application were both required dimensions.

Time-based user accounts In addition to the use of time in access rules, user accounts in HomeOS can have an associated time window of validity. This window is used to simplify guest access, which studies have shown to be both common and particularly problematic [22, 33]. Home owners can start access for a user at a certain time (e.g., for a future guest) and terminate access at a certain time (e.g., when the guest is expected to leave). The data corresponding to the guest (e.g., access privileges) are not deleted automatically after the validity window, to simplify reinstating access at a later time.

Hierarchical user and device groups Groups in HomeOS are arranged in a tree hierarchy. In contrast, groups in current OSes can be independent sets. We picked the tree organization because of its simplicity. When a user group is given access, it enables an easier determination of which users are given access. A user who is not part of this group will not inadvertently gain access because she is part of another group.

For devices, we use a tree hierarchy that is rooted in space because that matches how users think of resources in the home. It also aligns well with physical access as it is delineated by rooms. To our knowledge, current OSes do not support such device groups. We find that device groups simplify management; users can specify policies for groups rather than individual devices.

Orthogonal to spatial grouping, HomeOS has a high-security group. Users can deem certain devices as high-security to avoid accidental access to such devices and simplify the task of keeping the home network secure.

Applications are not given access to secure devices by default, and the user must explicitly provide access to such devices. We add some common classes of devices (e.g., cameras, locks) automatically to this group; users can later add or remove devices to or from this group.

Access control also forms the basis for privacy in our design. Applications cannot access sensor data unless they are granted access to those devices. Further, network access is disabled by default, so they cannot leak information externally. (Software updates are downloaded and applied by HomeOS.) Thus, we coarsely control privacy at the granularity of applications and devices. In the future, we will consider finer-grained control [42, 44].

3.4 Application layer

The application layer is where developer-written code runs. The key feature this layer provides, beyond the ability to use and compose devices, is the ability to determine if an application is compatible with the home and what services and/or devices are missing if it is not.

Today, users have little assurance that a given piece of software will work in their home. To address this uncertainty, HomeOS requires that applications provide a *manifest* describing what services they need. This enables it to determine if an application will function with the current device services in the home. (A similar approach is being used to manage handset diversity in smartphones today.) If the manifest indicates an application is not compatible, HomeOS can also determine what additional devices or services are needed.

A manifest has mandatory and optional features. Each feature is a set of roles, at least one of which is needed. For instance, an application may specify {"TV", "SonyTV"}, {"MediaServer"} as mandatory features, indicating that it needs a service with at least one of the two TV roles and a service that exports a media server. It might have {"Speaker"} as an optional feature if it offers enhanced functionality with that role.

Our current manifest descriptions cannot encode complex requirements (e.g., if an application needs devices to be in the same room). They handle what we deem to be the common case. Should the need arise, it would be straightforward for us to enhance manifest descriptions.

4 Design and implementation

HomeOS is an implementation of the above architecture as a component-based OS. All functionality that is not central to the platform is implemented by software components called *modules*. Modules that sit in the application layer are *applications*, and those that sit in the DCL and DFL are *drivers*.

4.1 Modules

Modules are the basic unit of functionality in HomeOS and, whether applications or drivers, they implement the API described in Figure 4(a).

Before a module can be run, it must be installed. Driver modules are installed when new devices are discovered on the home network; applications are installed in response to explicit user directives. Modules are installed by copying the binaries and accompanying metadata (e.g., manifests) to a specific directory. Installation is carried out only if the module is deemed compatible with the devices in the home. This check is also performed each time the application is run to deal with configuration changes. During installation, users specify if the module should be started automatically upon system (re)start or only upon explicit user request. Module updates and uninstallation are carried out by HomeOS and not by the module itself.

Running modules are isolated to prevent any interaction except via the APIs to the HomeOS platform and the service interface. By default modules are denied access to the network. DCL modules are the exception as some must use the network to control their associated devices. Even then, when possible we limit connectivity to only those devices. A module's file system access is limited to its own working directory where it can store its data and configuration.

HomeOS relies on DCL modules to discover new devices on the home network by running protocol-specific discovery protocols. (We also support a mode where users can also manually add a device if HomeOS is unable to do so automatically.) Once a new device is discovered, HomeOS installs a DFL module for it based on a database of device type to driver mappings. The device type is reported by the DCL module and is protocol-specific. The DFL module is granted access to the service that the DCL module exports for this device.

4.2 Services

Services are the only way that modules are allowed to interact with each other. They do so using a standardized API described in Figure 4(b). Modules advertise the services they offer to HomeOS which keeps a history of offered services to enable future compatibility testing.

Start:	Called to start a module; modules are garbage collected when it returns
Stop:	Called to request a module to stop; where state can be cleaned up before exit
SvcRegistered:	Called when a new service becomes active; used to listen for services of interest
SvcDeregistered:	Called when a service becomes in active; used to avoid using inactive services
AsyncReturn:	Called whenever a subscription generates an event or asynchronous call returns

(a) API for HomeOS modules

InitSvcAndCapability:	Creates a service and a capability to access it; returns the service handle back
RegisterSvc:	Registers the service as active advertising it to other modules
DeregisterSvc:	Marks a service as being inactive and notifies other modules
GetAllSvcs:	Returns a list of all active services
GetCapbility:	Requests a capability to access a given service
IsMySvc:	Returns whether a given service belongs to this module
Invoke:	Used to call an operation either synchronously or asynchronously
Subscribe:	Subscribe to notifications from an operation
SpawnSafeThread:	Create new thread which safely propagate its failures

(b) API For finding and interacting with HomeOS services

Figure 4: The APIs for modules (a) and services (b) in HomeOS

Modules inform HomeOS about services of interest to them using role names and/or locations. (Note that expressing interest does not grant access to a service other than to know of its existence and description.) When a service is registered, modules that have expressed interest in it are notified. A module can then query the service for its description as well as its location. Querying for the existence of a service does not require access privileges.

When a module needs to invoke an operation on a service, it requests a capability [31] from HomeOS. As part of the request, the module passes the credentials of the user it is running on behalf of. Without valid credentials, the request is successful only if the access is legal for all users. Drivers are handled slightly differently and typically have rules that give them access to their corresponding devices regardless of the user. User-controlled policy is applied when applications access the driver rather than when the driver accesses the device.

The legality of the requested access is evaluated by HomeOS based on the user, module, service, and time of day, by formulating the check as a Datalog query over the users table and access rules. If legal, a capability is generated and returned. A copy of the capability is then passed to the target service; this makes it easier for HomeOS to revoke the capability later if needed. Subsequently, the requesting module can use the capability to make calls directly to the service. HomeOS capabilities have an expiration time based on access rules.

An operation's callers must also include input parameters of the right type. HomeOS supports both primitive and complex types, which are passed by reference across isolation boundaries to avoid the overhead of serialization. Finally, operation invocations include a timeout. Unless the called service responds within this time limit, a timeout error code is returned.

4.3 HomeStore

To simplify the process of finding new applications and devices, inspired by smartphone app stores, HomeOS is coupled with HomeStore which hosts all HomeOS applications and drivers. It indexes application manifests as well as drivers' associated devices and exported services. It helps users find applications that are compatible with their homes, by matching manifests against services in their home. If an application is not compatible, it can recommend additional devices that meet the requirements.

4.4 Management tasks

To explain how users manage their homes using HomeOS we describe four important management tasks.

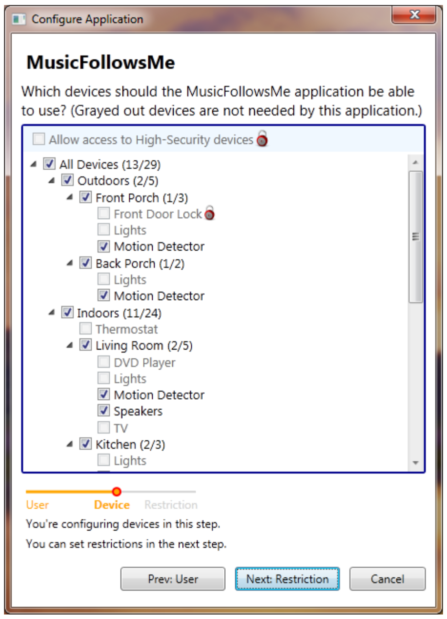
1. Adding a new application Users can browse applications and view their compatibility within HomeStore. Upon installing an application, HomeOS walks the user through setting up access control rules for the application. The core of this task is specifying which devices (or services[1]) the application should be allowed to access.

Since there may be hundreds of devices, we use the application manifest and service descriptions to show only compatible, non-secure services. Once the user selects which services the application can access, HomeOS uses the Datalog rule database to detect if the new application could access a device at the same time as other applications. If so, it asks the user which application should have a higher priority.

2. Adding a new device Once a new device is registered, users need to specify its location and whether it should be marked secure. They also need to configure which existing applications should have access to the device. This task is again simplified using application manifests, as only applications compatible with the new device are presented as valid options.

3. Verifying access rules To verify access control configuration, HomeOS allows users to view the rules from different vantage points using faceted browsing [25] (found on shopping Web sites to filter content

[1]HomeOS devices are exposed as services, as are features like notification and face recognition. To ease exposition, we refer to devices.


Configure Application

MusicFollowsMe

Which devices should the MusicFollowsMe application be able to use? (Grayed out devices are not needed by this application.)

- [ ] Allow access to High-Security devices
- [✓] All Devices (13/29)
  - [✓] Outdoors (2/5)
    - [✓] Front Porch (1/3)
      - [ ] Front Door Lock
      - [ ] Lights
      - [✓] Motion Detector
    - [✓] Back Porch (1/2)
      - [ ] Lights
      - [✓] Motion Detector
  - [✓] Indoors (11/24)
    - [ ] Thermostat
    - [✓] Living Room (2/5)
      - [ ] DVD Player
      - [ ] Lights
      - [✓] Motion Detector
      - [✓] Speakers
      - [ ] TV
    - [✓] Kitchen (2/3)
      - [ ] Lights

User Device Restriction

You're configuring devices in this step.
You can set restrictions in the next step.

Prev: User | Next: Restriction | Cancel

Figure 5: A GUI screen capture showing how applications are given access to devices

along multiple dimensions). This enables users to pose questions such as what devices an application can access or what devices can be accessed at night by a user group. The questions are answered using Datalog queries.

4. Adding new users When a new user account is added, the administrator must specify their group (e.g., guest) and the time window of account validity.

We have built a complete user interface (UI) to support these tasks. An example screenshot is shown in Figure 5 which occurs during the course of adding applications. We omit detailed description of the UI for space constraints, but note that it closely mirrors our system primitives and includes heuristics designed to minimize the exposure to risk even if users click OK repeatedly during configuration activities. Evaluation of its usability, which we discuss later, also evaluates the manageability of our primitives.

4.5 Implementation

We implemented HomeOS in C# using the .NET 4.0 Framework. We use the *System.AddIn* model which allows dynamic loading and unloading modules. It also offers module version control allowing the HomeOS kernel and individual modules to evolve independently. We isolate modules using *AppDomains*, a lightweight sandboxing mechanism [2]. Each module runs inside a domain. Direct manipulation is not allowed across domains. Instead, communication is done only through typed objects exchanged through defined entry points and subject to

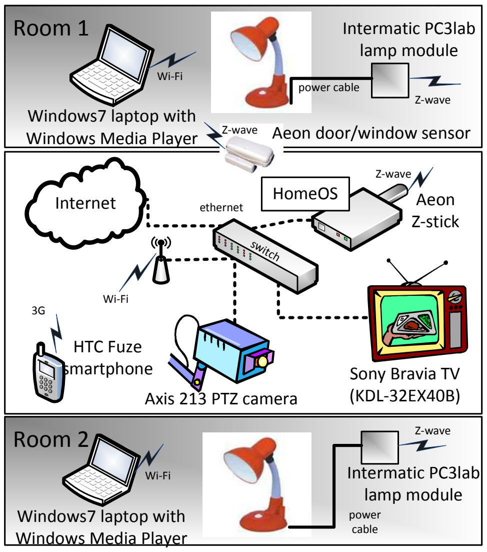


Figure 6: Our implementation testbed

access control. As our evaluation shows, the overhead of this isolation mechanism is low enough to support interactive applications. However, it does not provide performance or memory pressure isolation; these are subjects of future work. For Datalog query evaluation, we use Security Policy Assertion Language (SecPAL) [40], after modifying it to support time comparisons.

In addition to the HomeOS kernel, we developed drivers to interact with a diverse set of off-the-shelf devices that includes Z-Wave lighting controls, door/window sensors, smartphones, network cameras, TVs capable of receiving DLNA streams, Windows PCs, and IR (infra-red) transmitters. Figure 6 shows one of our testbeds with some of these devices.

Interoperability protocols greatly simplified the task of making these devices work with HomeOS because a module corresponding to the protocol can communicate with different devices. For instance, the same DLNA module works with both Windows 7 computers and the Sony TV; and the Z-Wave lightswitch module works with both Aeon and Intermatic devices. Greater adoption of interoperability protocols will make it easier to integrate devices with HomeOS. See below, however, for some shortcomings of current interoperability protocols.

We developed 18 applications that use these devices and run on our testbeds. While some applications require access to only one device (e.g., turn on or off a light), others are quite complex. For example, a media application transparently redirects a music stream from one room to another, depending on how lights are turned on or off in each room (using lights as a proxy human presence in the room). We also implemented a two-factor authentication application that triggers a configured action (e.g., open a lock) when the same person authenticates with their voice on the phone (speech recognition) and with their face on the camera (face recognition).

Each of the 18 applications is less than 300 lines of C# code and took only a few hours to develop. Because applications are written against high-level abstractions, most of the effort went toward application-specific logic. As we report below, other developers also found application development in HomeOS to be easy.

5 Experience

HomeOS currently runs in 12 real homes and 42 developers have written modules for it. These field experiences validate key aspects of the HomeOS architecture. They also evaluate the utility of the PC-like abstraction for home technology as well as situations in which HomeOS was more or less able to preserve that abstraction. Generally, the experiences were positive even for people with no prior home automation experiences. However, the experiences did reveal some rough edges of our prototype and limitations of current interoperability protocols.

We describe the main findings below, based on informal surveys of our users and developers. The next section presents results from controlled experiments.

5.1 Developers

We gave the HomeOS prototype to ten academic research groups, for use as a platform for both teaching and research on home applications. As part of this program, 42 undergraduate and graduate students developed tens of HomeOS applications and drivers.

They extended HomeOS in several directions [3, 4, 26, 27]. They wrote drivers for new devices including energy meters, different network cameras, appliance controllers and IM communication. They wrote new applications such as energy monitoring, remote surveillance, and reminders based on face recognition. The PC-like abstractions of drivers and applications enabled them to build software quickly and in reusable modules. Moreover, as a testament to the flexibility and extensibility of its architecture, we were not required to—and did not—modify HomeOS to support these development efforts.

As an example, one group extended HomeOS to support the Kinect RGB-D camera and built an application which allowed users to control lights via gestures (Figure 7). They were able to do this without having to wait for Kinect integration with a large commercial system (e.g., Control4) and were able to get it to interact

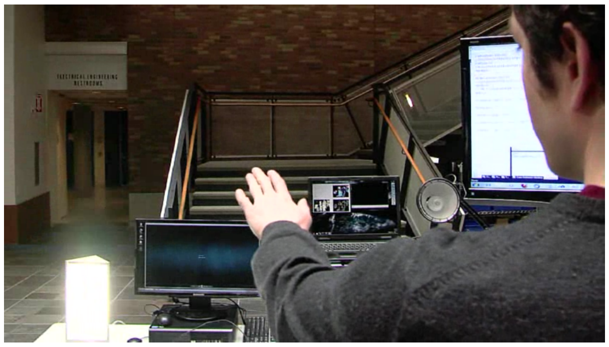


Figure 7: A student demonstrating how to turn on and off lights via gestures with a Kinect

with existing devices. Another group built an application that plays audio reminders based on who is recognized on cameras. This works with webcams, security cameras, Kinect or IP cameras and with any device HomeOS can play audio through (right now PCs, DLNA TVs, and Windows Phones). This underlines the power HomeOS gives to application developers to easily span multiple types of devices (security, PC, phone, entertainment, etc.). Commercial systems today support only a subset of devices related to their target scenario (e.g., security systems focus on cameras and motion sensors).

Layering and programmability Developers who wrote applications found the protocol independence of the APIs appealing. Developers who wrote new drivers for devices with existing DCL modules (e.g., a Z-Wave appliance controller) liked that they did not have to concern themselves with the low-level connectivity details and could instead focus exclusively on device semantics.

Interestingly, developers who extended HomeOS to devices without an existing DCL module (e.g., ENVI energy meters [19]) started by building one module that spanned both the DCL and DFL. For them, the split was unnecessary overhead as only one device used the connectivity protocol. However, in one case a group had to support multiple devices with the same connectivity protocol based on IP to Z-Wave translation. This group found value in separating functionality across two layers indicating it was not just an artifact or our experience.

Hardware-software coupling Our developers sometimes wanted to use device features that were not exposed to third-parties over the network. For instance, one developer wanted to insert a text notification on a TV without otherwise interrupting the on-screen video. Today, some set-top boxes have this capability (e.g., for cable TV operators to signal caller identity of incoming calls), but they do not expose it to third-party software.

This points to an inherent advantage of vertically integrated software—being able to better exploit device capabilities—that open systems like HomeOS lack. This is unsurprising in retrospect as the closed nature of current solutions and devices is what HomeOS attempts to combat. However, the systematic way vendors can expose device capabilities in HomeOS should encourage them to make their capabilities available to applications.

Media applications and decentralized data plane A few developers had difficulty in writing media applications. HomeOS centralizes the control plane but not the data plane to avoid creating a performance bottleneck. If two devices use the same protocol, we assume that they can directly exchange data. Thus, we assume that a DLNA renderer can get data directly from a DLNA server once provided with a media URI. The DLNA protocol turns out to not guarantee this because of video encoding and/or resolution incompatibilities. While we currently use heuristics to provide compatible formats when transcoding is available, they are not perfect.

For reliable operation, we also plan to use HomeOS as a transcoding relay (thus, centralizing the data plane and more closely mirroring the PC abstraction) when data plane compatibility between nodes is not guaranteed. As high-quality open source transcoders exist [21], the main technical challenge is to generate profiles of what input and output formats devices support. This requires parsing device protocols like DLNA. Although this means violating HomeOS's agnostic kernel, we believe that media applications are common and important enough to justify an exception.

5.2 Users

Twelve homes have been running HomeOS for 4–8 months. We did not actively recruit homes but many approached us after becoming aware of the system. We limited our initial deployment to 12 homes. Ten of them had no prior experience with home automation. Beyond providing the software and documentation, we did not assist users in running or managing HomeOS. These homes are using a range of devices including network cameras, webcams, appliance and light controllers, motion sensors, door-window sensors and media servers and renderers.

Organic growth What our users found most attractive was being able to start small and then expand the system themselves as desired. At first, they typically did not know what they wanted and only discovered what they found valuable over time. HomeOS let them start small (at low cost) and extend as needed. It thus provided a system that was much more approachable than commercial systems today that require thousands of dollars upfront. It was also more likely to satisfy users by allowing them to evolve it to meet their needs rather than requiring them to make all decisions during initial installation [7].

Indeed, all users employed an organic growth strategy. One user started running HomeOS with only one network camera to view his front yard on a smartphone while away from home. He later added two more cameras—a webcam and a network camera from a different vendor—and was able to continue using the same application without modification. He then added two sensors to detect when doors were opened so that he could be notified when unexpected activity occurred. This used our door-window monitoring application sends email notifications, which can contain images from any cameras in the home. The user later added two light controllers and another application to control them. What started off as simply wanting to see his front yard from work evolved into a notification system and lighting control.

Diagnostic support in interoperability protocols On the negative side, at least two homes had problems diagnosing their deployments. For instance, when applications that use Z-Wave devices behaved unexpectedly, users could not easily tell if it was due to code bugs, device malfunctions or poor signal strength to the device. Disambiguation requires effort and technical expertise (e.g., unmount the device, bring it close to the controller, and then observe application behavior).

This difficulty is an instance where the added complexity of network devices, in contrast to directly connected peripherals, becomes apparent. Countering it requires diagnostic tools but they are hard to build today because interoperability protocols have limited diagnostic support. We thus recommend that device protocols be extended to provide diagnostic information. Even something akin to ICMP would be a step forward.

6 Evaluation

In addition to our experience with real homes and developers, we evaluate HomeOS through controlled experiments, focusing on its ease of programming, ease of managing and system performance. This gives us quantitative validation to confirm our real-world experiences above. We find that developers can write realistic applications within 2 hours, that users can use our management interfaces with similar success to other carefully designed systems and that system performance is good enough to easily support rich, interactive applications.

6.1 Ease of programming

To evaluate how easy it is to write a HomeOS application, we conducted a study where we recruited student and researcher volunteers to develop HomeOS applications. (Different from the students mentioned in §5.)

We provided our participants with a brief introduction to HomeOS, some basic documentation on our abstractions, all the drivers, and four simple applications that use only one driver each—image recognition, camera snapshot, DLNA music player, and light-switch controller. Each participant got a total of five-minutes of verbal instructions (with no demonstration of code) on the goal of the study and pointers to these resources. We left the participant and the testbed, with the HomeOS server console running an IDE (Visual Studio) configured to use HomeOS binaries. We provided little assistance beyond the initial training, though on three occasions the participants uncovered bugs in our system that we had to fix before they could proceed.

We gave each participant the task of writing one of two applications for our testbed. "Custom Lights Per-User" (CLU) will adjust the lights in any room based on its occupant's preferences. This application needs to find cameras in the house to which it has access, continuously poll them, use the image recognition service to identify the occupant (if any), and set the "dimmers" in the camera's room to the occupant's preferences. For testing, we gave each participant two photographs on which the image recognition service was trained and the user-to-lighting preference chart.

The second application—"Music Follows the Lights" (MFL)—was one we previously built but did not provide to participants. This application finds all lights and media devices in the house, registers for changes to the lights' status and plays music (from a media server) on an audio device in rooms with lights that are on.

We recruited ten participants for this study via a mailing list within our organization. Seven were graduate students and three were researchers. Only one had prior experience with home automation, and none had significant prior experience with programming service-based abstractions (e.g., WSDL or SOAP). This level of expertise is at the low-end of what we expect of future HomeOS developers. We gave each participant two hours to write an application. Half of the participants were given the first application and half were given the second.

Figure 8(a) summarizes the results of our study. The time reported was computed from the end of verbal instructions until the participant was done, minus any breaks the participant took and the time we spent correcting bugs. Eight participants developed complete applications within approximately two hours (126 minutes). Of the two who did not finish, one spent the bulk of the time developing a "slick GUI" for the application instead of its core logic and the other did not realize that HomeOS drivers were not running by default. This problem stemmed from a misinterpretation of instructions and could have been avoided with clearer instructions.

In the exit interview, almost all participants (even those who did not finish) reported that HomeOS was "very programmable" and the APIs were "natural." However, they also expected more syntactic support in the IDE for invoking operations. We are addressing this issue by defining a C# interface for each role.

These results suggest that it is easy to develop applications for HomeOS. Even without prior experience, developers were able to implement realistic applications in just a couple of hours. We do not mean to suggest that all HomeOS applications can be developed in two hours. Our study emphasized the use of HomeOS's basic abstractions and did not require the developers to focus on consumer-facing issues such as a richer GUI. However, it does provide evidence that the base programming abstractions are a good fit for applications in the home.

6.2 Ease of managing

Our second study evaluates whether our management primitives and interfaces are easy enough for non-expert home users to use. We find that with no training, typical home users are able to complete typical management tasks correctly around 80% of the time.

Methodology We began each session by explaining the background and goals of the study and the three security principals—users, devices and applications. We asked participants to pretend to be a member of the following imaginary family. Jeff and Amy are husband and wife. Dave and Rob are their eight-year and seven-month old kids. Jeff's brother Sam, who visits occasionally, has a guest account. The house has 29 devices of nine different types. Three of the devices—camera and microphone in Rob's room, and the front door lock—are high-security. The family has four applications for lighting, monitoring, and temperature control, and fourteen rules specifying access controls policies. We assigned the male participants to play Jeff and females to play Amy.

We then asked them to perform the 7 management tasks show in Figure 8(b) using our UI. These tasks reflect what we expect users to do with HomeOS and span key management tasks (§4.4). Tasks 1, 2 and 6 require configuring applications, including restricting their use to certain users, devices and times of day. Task 3 requires configuring a new device with group and application access. Task 4 requires adding a new guest. Tasks 5 and 7 require verifying policies based on specific concerns.

At the same time, our tasks stress the ability of primitives in HomeOS to simplify management. For instance, Tasks 1 and 2 use application manifests, Datalog rules,

	app	LoC	mins
1	CLU	183	84
2	CLU	193	62
3	CLU	156	66
4	CLU	172	113
5	CLU	221	107
6	MFL	224	95
7	MFL	244	126
8	MFL	239	102
9	MFL	303	93*
10	MFL	130	100*

(a) Programming study

Task	✓
1. Configure your new EcoMonitor app. Let it access all but high-security devices for everyone.	11
2. Configure your new MusicFollowsMe app. Let it access all motion detectors and speakers but no high-security devices. All residents can use it but kids cannot not play music in the parents' bedroom	9
3. Configure your new kitchen security camera. Mark it high-security and let HomeMonitor access it.	11
4. Give guest access to Jane, who will be visiting until September 6th. Place her in the Guests group so that she can use appropriate apps during her visit.	12
5. Check the rules and tell the facilitator which apps can access high security devices.	1
6. Configure your new OpenFrontDoor app. Residents can use it any time. Sam (guest) cannot use it at all. Jane (guest) can use it only during the day (8 AM to 8 PM).	11
7. Check if only adults can access the camera in Rob's room and only using HomeMonitor.	10

(b) Management study

Figure 8: Results of our two studies: (a) The application developed and time taken by each participant, '*' implies an incomplete program; (b) Assigned management tasks and the number of participants (of 12) that completed each accurately

and user and device groups, whereas Task 4 uses time-based user accounts and user groups. Ideally, we would evaluate each primitive separately but we found management tasks that stress only one primitive to be unrealistic.

We simulated a real-world setting by not training our participants in using the technology they are required to manage. Instead we provided manuals for each task type and told them that reading the manuals was not required but they could refer to them anytime if they wanted.

Participants We had twelve participants (eight male, four female) from the Greater Puget Sound region in Washington state. We recruited the participants through a professional recruiting service. We screened them to ensure they are somewhat familiar with home technology. They were required to own at least one TV, one computer, and three types of electronic devices (e.g., wireless router, security camera, smartphone, etc.). They were also required to be able to conduct basic administrative tasks (e.g., set up an account on a computer)[2].

Results Figure 8(b) shows the number of participants that completed each task correctly. We see that the accuracy rate is high. Overall, it is 77%; ignoring Task 5 (discussed below), it is 89%. This result is encouraging because it was obtained without any training and many participants did not use the manual. For reference, we note that our accuracy rate is similar to that obtained with careful design of system semantics and interfaces for file system access control [38] and firewall configuration [37]. Our participants took one to four minutes to complete individual tasks.

While the errors in most tasks were simply forgetting a single click, Task 5 was particularly problematic. Only one participant was able to complete it correctly. Others had difficulty forming the correct query for the task using our faceted browsing interface. (Forming the query for Task 7, which also used the same interface, was not as problematic.) They could correctly and easily tell that the HomeMonitor application was using the camera, but did not realize that so was another application. We plan to address this by augmenting the UI to reduce the work needed to detect things that are unexpected or the absence of use by other applications. We believe that the underlying primitives do not need modification.

In the exit interview, we asked the participants how easy HomeOS was to use and learn, on a 7-pt. Likert scale from "Strongly Disagree" (1) to "Strongly Agree" (7). The participants found the system easy to learn (avg. 6.0), easy to use (avg. 6.0), and intuitive (avg. 5.5).

6.3 System performance

In addition to easy programming and management, HomeOS must have acceptable performance. Our goals are to have latency that is low enough to run responsive, interactive applications and to offer scalability and throughput that can handle large, complex homes. To quantify the overhead of layering in HomeOS, we compare against a hypothetical, monolithic system without layering and isolation.

Experimental setup To gather performance data about HomeOS, we ran a simple benchmarking application using a virtual device on a quad-core Intel Xeon 2.67 GHz PC. Unless otherwise specified, there is one application and one driver running. The application generates load by creating ten continuous tasks that attempt to invoke an operation on the device at a fixed rate, but are scheduled by .NET ThreadPool which dynamically picks a number of threads to execute based on current performance.

Latency of operation invocation Figure 9 shows the latency of an operation invocation with no arguments under different loads. While we incur higher overhead than without isolation, the difference is approximately only 25% or a few hundred microseconds. Even under heavy load, we are able to keep latencies below 2 ms. This is two orders of magnitude lower than the interactive response time guideline of 100 ms [18], which means that

[2]While this may seem to exclude typical home users, homes typically have at least one experienced tech guru [36].

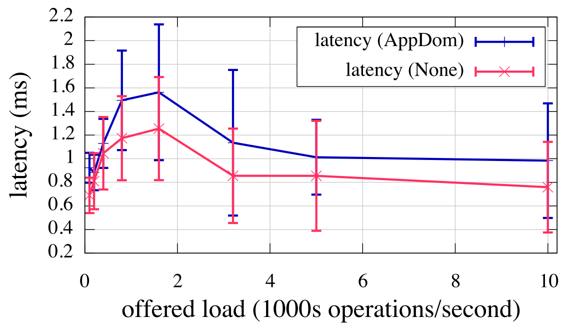


Figure 9: Operation invocation latency as a function of offered load with and without Application Domain isolation

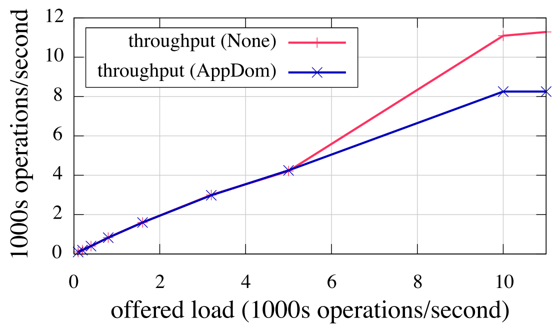


Figure 10: Operation invocation throughput as a function of offered load with and without Application Domain isolation

applications can compose several services and allow for network delays before responding to the user.

The odd increase in latency when the offered load is low (between 100 and 3000 operations per second, respectively) is an artifact of the ThreadPool scheduling rapidly switching between threads when each thread offers substantially lower load than what the machine can handle. We found it to persist across other microbenchmarks as well.

Throughput of operations To evaluate the load that HomeOS can handle, we tracked the throughput of the system at different offered loads both with and without AppDomain isolation. Figure 10 shows that the throughput of the two modes mirror each other until the system is driven near peak throughput. With AppDomain isolation, HomeOS handles approximately 8,250 operation invocations per second, while with no isolation the system can handle nearly 11,300 operation invocations per second. Beyond that load, ThreadPool scheduling backs off. If an application does not use this or a similar mechanism, latencies climb substantially when the system is driven past the load it can handle as one would expect. This level of performance has been well-beyond what was required for any of our current deployments.

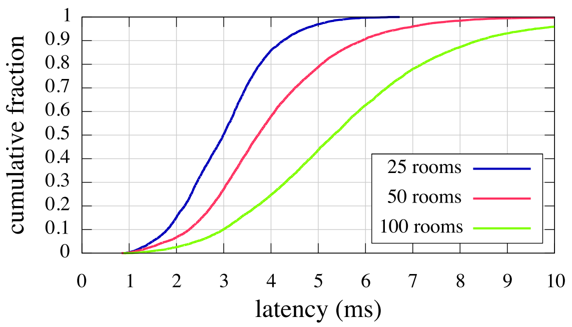


Figure 11: Cumulative fraction of operation invocations completed within a given latency for varying numbers of applications and devices

Scalability To understand how HomeOS scales to a large home with many devices and applications, we examine the latency of operation invocation in an extreme setting. We emulate a large home with a varying number of rooms, each room containing 4 devices and one highly-active application querying each device 10 times per second. Figure 11 shows the operation invocation latency in this setting. As we increased the number of rooms from 25 to 50 to 100 (1,000, 2,000 and 4,000 total operations per second), we see a median latency of 3–5.5 ms depending on the number of applications. The latency in this experiment is higher than those presented earlier at the same total load because of the added overhead of having 125–500 modules running rather than two. The threads across modules are not managed as part of the same ThreadPool. Despite this additional overhead, the vast majority of operations complete within 10 ms, which is an order of magnitude below the interactive guideline of 100 ms. This result suggests that HomeOS can easily scale to large, well-connected homes.

7 Related work

While we draw on many strands of existing work, we are unaware of a system similar to HomeOS that provides a PC abstraction for home technology to simplify management and application development while remaining extensible. We categorize related work into five groups.

1. Device interoperability Many systems and standards for providing device-to-device interoperability in the home exist. They include DLNA [14], UPnP [41], Z-Wave [46], ZigBee [45], and Speakeasy [16]. HomeOS is agnostic to what interoperability standards are used and can incorporate any of these. But, as discussed earlier, while interoperability is helpful, it does not provide enough support for users and developers.

2. Multi-device systems Many commercial home automation and security systems integrate multiple devices in the homes. Like HomeOS, they centralize control, but such systems tend to be monolithic and hard for users to extend. For instance, Control4 [11]—one of the most extensible automation systems—allows for only its own devices and a limited set of ZigBee devices; and offers only a limited form of programming based on rules of the form "upon Event E, do Task T." Further, the technical complexity of installing and configuring Control4 and other systems (e.g., HomeSeer [28], Elk M1 [17] and Leviton [30]) can be handled only by professional installers (which is expensive) or expert hobbyists. In the research community, EasyLiving [29] was a monolithic system with a fixed set of applications integrated into the platform. In contrast to such systems, we focus on building a system that can be extended easily with new devices and applications by non-experts.

3. Programmability for the home We proposed the idea of a home-wide OS in a position paper [13]. This paper presents an architecture based on our experiences, a more complete system, and its evaluation.

Newman proposes using "recipes," [34] which are programs in a domain-specific language that compose devices in the home. He also advocates using marketplaces to disseminate recipes. With HomeOS applications, our vision is similar. Newman does not discuss how recipes can be realized in practice, which requires tackling challenges similar to those we address.

Previous work also advocates using a central controller to simplify integration [20, 39]. They have a different scope than HomeOS. For instance, Rosen et al. [39] (incidentally, also called HomeOS), focus on providing context such as user location to applications. These works offer little detail about their design and implementation.

Other systems have employed services in the home environment. iCrafter is a system for UI programmability [35]. ubiHome aims to program ubiquitous computing devices inside the home using Web services [24]. While our use of the service abstraction is similar to these systems, we engineer a more complete system for programming and managing devices in the home.

4. Management challenges in the home Calvert et al. outline the various management challenges in the home network [8], and like us, argue for centralization. We go beyond management issues and also focus on simplifying application development. Further, in contrast to their proposal, we do not require device modifications.

5. OSes for network devices Researchers have designed OSes over multiple devices in other domains. iROS aims to simplify programming devices such as displays and white-boards in collaborative workspaces [6]. NOX aims to simplify managing switches in enterprise networks [23]. While conceptually similar, HomeOS handles complexities specific to the home environment.

8 Conclusions and future work

HomeOS simplifies the task of managing and extending technology in the home by providing a PC-like abstraction for network devices to users and developers. Its design is based on management primitives that map to how users want to manage their homes, protocol-independent services that provide simple APIs to applications and a kernel that is agnostic of the functionality and protocols of specific devices. Experience with real users and developers, in addition to controlled experiments, help validate the usefulness of the abstraction and our design.

This experience also reveals gaps where we could not cleanly implement the abstraction due to limitations of device protocols (e.g., little support for diagnosis and incompatible implementations across vendors) or due to limited features being exposed by devices over the network. We plan to address these limitations in the future.

More broadly, our hope is that this work spurs the research community to further explore the home as a future computing platform. While we cannot outline a complete agenda for work in this area, we point out two fruitful directions based on our experience:

1. Foundational services Over the years PC applications have come to expect some essential services that the OS provides (e.g., a file system). Are there similar services for the home environment? Such services should not only be broadly useful but also almost universally implementable. For instance, consider occupancy information—which rooms are currently occupied by people. It can benefit many applications (e.g., lighting control, thermostat control, and security), but depending on the devices in the home, it may be difficult to infer reliably (e.g., motion sensors can be triggered by pets; cameras are more reliable). Making occupancy an essential service requires each home to possess the necessary devices, thus increasing the cost of a basic HomeOS installation. (This is akin to PC or smartphone OSes specifying minimum hardware requirements.) Thus, careful consideration is needed to determine which services a system like HomeOS should provide in all homes.

2. Identity inference Some desired reactions to physical actions in the home depend on the identity of the user or who else is around. For instance, users may want to play different music based on who entered and turned on the lights, or parents may not want their children to turn on the Xbox in their absence. Currently, HomeOS can either not support such policies (lightswitches have no interface to query user identity) or support them in an inconvenient manner (ask parents for their password). A promising avenue for future work is to build non-intrusive identity inference (e.g., using cameras in the home, or users' smartphones), and then allow users to express policies based on that inference. A key challenge in realizing this system is to maintain safety in the face of possible errors in identify inference.

Acknowledgements Tom Anderson, S. Keshav, Ed Nightingale, Andrew Warfield and our anonymous reviewers provided helpful comments on this paper. The volunteers who were brave enough to run HomeOS in their homes, the students who developed HomeOS applications and drivers, and our study participants provided valuable feedback that helped refine our design. Frank Martinez was our first volunteer and helped set up the volunteer program. We are grateful to them all.

References

[1] Home security systems, home security products, home alarm systems - ADT. http://www.adt.com.

[2] Application Domains. http://msdn.microsoft.com/en-us/library/2bh4z9hs%28v=vs.100%29.aspx.

[3] O. Ardakanian, S. Keshav, and C. Rosenberg. Markovian Models for Home Electricity Consumption. In *SIGCOMM Workshop on Green Networking*, 2011.

[4] N. Banerjee, S. Rollins, and K. Moran. Automating Energy Management in Green Homes. In *SIGCOMM Workshop on Home Networks (HomeNets)*, 2011.

[5] L. Bauer, L. Cranor, R. W. Reeder, M. K. Reiter, and K. Vaniea. A user study of policy creation in a flexible access-control system. In *CHI*, 2008.

[6] J. Borchers, M. Ringel, J. Tyler, and A. Fox. Stanford Interactive Workspaces: A Framework for Physical and Graphical User Interface Prototyping. *IEEE Wireless Communications. Special Issue on Smart Homes*, 2002.

[7] A. J. Brush, B. Lee, R. Mahajan, S. Agarwal, S. Saroiu, and C. Dixon. Home Automation in the Wild: Challenges and Opportunities. In *CHI*, 2011.

[8] K. L. Calvert, W. K. Edwards, and R. E. Grinter. Moving Toward the Middle: The Case Against the End-to-End Argument in home networking. In *HotNets*, 2007.

[9] S. Ceri, G. Gottlob, and L. Tanca. What you Always Wanted to Know About Datalog (And Never Dared to Ask). *IEEE Transactions on Knowledge and Data Engineering*, 1, 1989.

[10] A. Chaudhuri, P. Naldurg, G. Ramalingam, S. Rajamani, and L. Velaga. EON: Modeling and Analyzing Access Control Systems with Logic Programs. In *CCS*, 2008.

[11] Control4 Home Automation and Control. http://www.control4.com.

[12] Crestron Electronic: Home automation, building and campus control. http://www.crestron.com.

[13] C. Dixon, R. Mahajan, S. Agarwal, A. J. Brush, B. Lee, S. Saroiu, and V. Bahl. The home needs an operating system (and an app store). In *HotNets*, 2010.

[14] DLNA. http://www.dlna.org/home.

[15] W. K. Edwards, R. E. Grinter, R. Mahajan, and D. Wetherall. Advancing the state of home networking. *Communications of the ACM*, 54, 2011.

[16] W. K. Edwards, M. W. Newman, J. Z. Sedivy, T. F. Smith, D. Balfanz, D. K. Smetters, H. C. Wong, and S. Izadi. Using SpeakEasy for ad hoc peer-to-peer collaboration. In *CSCW*, 2002.

[17] M1 Security & Automation Controls. http://www.elkproducts.com/m1_controls.html.

[18] Y. Endo, Z. Wang, J. B. Chen, and M. Seltzer. Using latency to evaluate interactive system performance. In *OSDI*, 1996.

[19] ENVI whole home energy monitor by powersave. http://www.currentcost.net.

[20] C. Escoffier, J. Bourcier, P. Lalanda, and J. Yu. Towards a home application server. In *Consumer Communication & Networking Conference*, 2008.

[21] FFmpeg. http://ffmpeg.org.

[22] R. E. Grinter, W. K. Edwards, M. Chetty, E. S. Poole, J.-Y. Sung, J. Yang, A. Crabtree, P. Tolmie, T. Rodden, C. Greenhalgh, and S. Benford. The ins and outs of home networking: The case for useful and usable domestic networking. *ToCHI*, 16(2), 2009.

[23] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker. NOX: towards an operating system for networks. *SIGCOMM CCR*, 38(3), 2008.

[24] Y.-G. Ha, J.-C. Sohn, and Y.-J. Cho. ubiHome: An Infrastructure for Ubiquitous Home Network Services. In *IEEE International Symposium on Consumer Electronics*, 2007.

[25] M. Hearst, A. Elliott, J. English, R. Sinha, K. Swearingen, and K.-P. Yee. Finding the flow in web site search. *Communications of the ACM*, 45(9), 2002.

[26] HomeOS. http://homeos.codeplex.com.

[27] HomeOS demos. http://research.microsoft.com/en-us/um/redmond/projects/homeos/homeos-demos.htm.

[28] Home Automation Systems - HomeSeer. http://www.homeseer.com.

[29] J. Krumm, S. Harris, B. Meyers, B. Brumitt, M. Hale, and S. Shafer. Multi-Camera Multi-Person Tracking for EasyLiving. In *IEEE Workshop on Visual Surveillance*, 2000.

[30] Leviton Online Store - LevitonProducts.com. http://www.levitonproducts.com.

[31] H. M. Levy. *Capability Based Computer Systems*. Digital Press, 1984.

[32] N. Li and J. C. Mitchell. Datalog with constraints: A foundation for trust management languages. In *International Symposium on Practical Aspects of Declarative Languages*, 2003.

[33] M. L. Mazurek, J. Arsenault, J. Breese, N. Gupta, I. Ion, C. Johns, D. Lee, Y. Liang, J. Olsen, B. Salmon, R. Shay, K. Vaniea, L. Bauer, L. F. Cranor, G. R. Ganger, and M. K. Reiter. Access Control for Home Data Sharing: Attitudes, Needs and Practices. In *CHI*, 2010.

[34] M. W. Newman. Now we're cooking: Recipes for end-user service composition in the digital home. In *IT@Home: Workshop associated with CHI*, 2006.

[35] S. R. Ponnekanti, B. Lee, A. Fox, P. Hanrahan, and T. Winograd. ICrafter: A Service Framework for Ubiquitous Computing Environments. In *Ubicomp*, 2001.

[36] E. S. Poole, M. Chetty, R. E. Grinter, and W. K. Edwards. More than Meets the Eye: Transforming the User Experience of Home Network Management. *Designing Interactive Systems*, 2008.

[37] F. Raja, K. Hawkey, and K. Beznosov. Revealing hidden context: improving mental models of personal firewall users. In *Symposium on Usable Privacy and Security (SOUPS)*, 2009.

[38] R. W. Reeder, L. Bauer, L. F. Cranor, M. K. Reiter, and K. Vaniea. More than skin deep: measuring effects of the underlying model on access-control system usability. In *CHI*, 2011.

[39] N. Rosen, R. Sattar, R. W. Linderman, R. Simha, and B. Narahari. HomeOS: Context-Aware Home Connectivity. In *International Conference on Pervasive Computing and Applications*, 2004.

[40] Security Policy Assertion Language implementation for Microsoft .NET. http://research.microsoft.com/secpal.

[41] Universal Plug-and-Play. http://www.upnp.org.

[42] S. VanDeBogart, P. Efstathopoulos, E. Kohler, M. Krohn, C. Frey, D. Ziegler, F. Kaashoek, R. Morris, and D. Mazieres. Labels and Event Processes in the Asbestos Operating System. *TOCS*, 25(4), 2007.

[43] J. Wu, A. Osuntogun, T. Choudhury, M. Philipose, and J. M. Rehg. A Scalable Approach to Activity Recognition based on Object Use. In *International Conference on Computer Vision*, 2007.

[44] N. Zeldovich, S. Boyd-Wickizer, and D. Mazieres. Securing distributed systems with information flow control. In *NSDI*, 2008.

[45] ZigBee Alliance. http://www.zigbee.org.

[46] Z-Wave.com - ZwaveStart. http://www.z-wave.com.

Structured Comparative Analysis of Systems Logs to Diagnose Performance Problems

Karthik Nagaraj   Charles Killian   Jennifer Neville

Purdue University

{*knagara, ckillian, neville*}@*cs.purdue.edu*

Abstract

Diagnosis and correction of performance issues in modern, large-scale distributed systems can be a daunting task, since a single developer is unlikely to be familiar with the entire system and it is hard to characterize the behavior of a software system without completely understanding its internal components. This paper describes DISTALYZER, an automated tool to support developer investigation of performance issues in distributed systems. We aim to leverage the vast log data available from large scale systems, while reducing the level of knowledge required for a developer to use our tool. Specifically, given two sets of logs, one with good and one with bad performance, DISTALYZER uses *machine learning* techniques to compare system behaviors extracted from the logs and automatically infer the strongest associations between system components and performance. The tool outputs a set of inter-related event occurrences and variable values that exhibit the largest divergence across the logs sets and most directly affect the overall performance of the system. These patterns are presented to the developer for inspection, to help them understand which system component(s) likely contain the *root cause* of the observed performance issue, thus alleviating the need for many human hours of manual inspection. We demonstrate the generality and effectiveness of DISTALYZER on three real distributed systems by showing how it discovers and highlights the root cause of six performance issues across the systems. DISTALYZER has broad applicability to other systems since it is dependent only on the logs for input, and not on the source code.

1 Introduction

Modern, large-scale distributed systems are extremely complex, not only because the software in each node in the distributed system is complex, but because interactions between nodes occur asynchronously, network message delays and orderings are unpredictable, and nodes are in heterogeneous environments, uncoupled from each other and generally unreliable. Compounding this is the fact that the software is typically developed by teams of programmers, often relying on external components and libraries developed independently, such that generally no one developer is fully aware of the complex interactions of the many components of the software.

Transmission [35] and HBase [20] exemplify the scale of this type of software development. Transmission is an open-source implementation of BitTorrent. In 2008, after three years of development, it became the default BitTorrent client for Ubuntu and Fedora, the two most popular Linux distributions. In the last two years alone, 15 developers committed changes to the codebase, not counting patches/bugs submitted by external developers. HBase is an open-source implementation of BigTable [5], depending on the Hadoop [19] implementation of the Google File System [17]. HBase has grown very popular and is in production use at Facebook, Yahoo!, StumbleUpon, and Twitter. HBase's subversion repository has over a million revisions, with 21 developers from multiple companies contributing over the last two years.

Given the activity in these projects, it not surprising that, in our experiments, we observed performance problems, despite their mature status. In systems with many independent developers, large user-bases with differing commercial interests, and a long history, diagnosis and correction of performance issues can be a daunting task—since no one developer is likely to be completely familiar with the entire system. In the absence of clear error conditions, manual inspection of undesirable behaviors remains a primary approach, but is limited by the experience of the tester—a developer is more likely to ignore occasional undesirable behavior if they do not have intimate knowledge of the responsible subsystems.

Recent research on distributed systems has produced several methods to aid debugging of these complex systems, such as execution tracing [13, 16, 31], replay debugging [15], model checking [23, 24, 29], live property testing [25], and execution steering [38]. However, these methods either require either extensive manual effort, or are automated search techniques focused on discovering specific *error* conditions.

To address the challenge of debugging undesirable behaviors (i.e., *performance* issues), we focus on comparing a set of baseline logs with acceptable performance to another set with unacceptable behavior. This approach aims to leverage the vast log data available from complex, large scale systems, while reducing the level of knowledge required for a developer to use our tool. The state-of-the-art in debugging the performance of request flows [2, 4, 6, 32] also utilizes log data; however, in contrast with this previous work, we focus on analyzing a wider range of system behaviors extracted from logs. This has enabled us to develop an analysis tool applicable to more than simply request processing applications. Other work in identifying problems in distributed systems from logs [37] is restricted to identifying anomalous local problems, while we believe that poor performance commonly manifests from larger implementation issues.

We present DISTALYZER, a tool to analyze logs of distributed systems automatically through comparison and identify components causing degraded performance. More specifically, given two sets of logs with differing performance (that were expected to have equivalent performance), DISTALYZER outputs a summary of event occurrences and variable values that (i) most diverge across the sets of logs, and (ii) most affect *overall* system performance. DISTALYZER uses *machine learning* techniques to automatically infer the strongest associations between system components and performance. Contributions of this paper include:

- An assistive tool, DISTALYZER, for the developer to investigate performance variations in distributed systems, requiring minimal additional log statements and post processing.
- A novel algorithm for automatically analyzing system behavior, identifying statistical dependencies, and highlighting a set of interrelated components likely to explain poor performance. In addition to the highlighted results, DISTALYZER also provides interactive exploration of the extended analysis.
- A successful demonstration of the application of DISTALYZER to three popular, large scale distributed systems–TritonSort [30], HBase & Transmission–identifying the root causes of six performance problems. In TritonSort, we analyzed a recently identified performance variation—the TritonSort developers surmised DISTALYZER could have saved them 1.5 days of debugging time. In follow-up experiments on Transmission and HBase, once we fixed the identified problems, their performance was boosted by 45% and 25% respectively.

2 Instrumentation

DISTALYZER derives its analysis based on the data extracted from logs of distributed systems executions. Hence, we describe the process of obtaining and preparing the logs for analysis, before the actual design in § 3. Applying our modeling to the logs of systems requires that some amount of its meaning is provided to DISTALYZER. Inherently, this is because we are not seeking to provide natural language processing, but instead to analyze the structure the logs represent. Xu *et al.* [37] have considered the automatic matching of log statements to source code, which requires tight coupling with programming languages to construct abstract syntax trees. In contrast, DISTALYZER aims to stay agnostic to the source code by abstracting the useful information in the logs. We describe this in more detail below.

The contributions of logging to debugging are so deeply ingrained that systems typically are not successful without a significant amount of effort expended in logging infrastructures. DISTALYZER assumes that the collection of logs has not affected the performance behaviors of interest in the system. This is a standard problem with logging, requiring developers to spend much effort toward efficient logging infrastructures. Logging infrastructures range from free text loggers like *log4j* [26], to fully structured and meaningful logs such as Pip [31] and XTrace [13]. Unfortunately, the common denominator across logging infrastructures is not a precise structure indicating the meaning of the logs.

Consider Pip [31], a logging infrastructure which provides log annotations indicating the beginning and ending of a task, sending and receiving of messages, and a separate log just as an FYI (a kind of catch-all log). Every log also indicates a path identifier that the log belongs to, thus it is possible to construct path trees showing dependencies between tasks within paths. This kind of instrumentation has been leveraged by Sambasivan *et al.* [32] to compare the path trees in systems logs. Unfortunately, this detail of logging is neither sufficient (it does not capture the instances of value logging, and does not adequately handle tasks which belong to multiple flows), nor is it widely available. A more commonly used logging infrastructure, *log4j*, provides a much more basic scheme - logs are associated with a "type," timestamp, priority, and free text string. It then remains to the developer to make sense of the logs, commonly using a brittle set of log-processing scripts.

As a compromise between fully meaningful logs and free-text logs, we work to find a middle-ground, which can be applied to existing logs without onerous modifications to the system being investigated. Our insight is that logs generally serve one of two purposes: *event log messages* and *state log messages*.

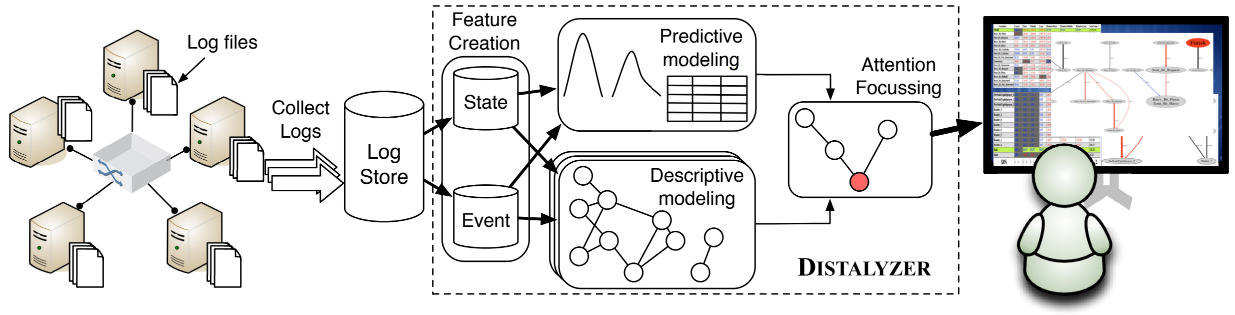


Figure 1: Four-step log comparison process in DISTALYZER leading up to a visual interface

Event log message. An event log message indicates that some event happened at the *time* the log message was generated. Most of the logging for Pip falls into this category, in particular the start or end of tasks or messages. Other examples of such logs include logs that a particular method was called, branch of code taken, etc. These logs are often most helpful for tracing the flow of control with time between different components of the system.

State log message. A state log message indicates that at the time the log message was generated, the *value* of some system variable is as recorded. Typically, a state log message does not imply that the value just became the particular value (that would instead be an event log message), but merely that at the present time it holds the given value. State log messages are often printed out by periodically executed code, or as debugging output called from several places in the code. State log messages are often most helpful for capturing snapshots of system state to develop a picture of the evolution of a system.

Distinguishing state and event log messages is an important step, that allows us to tailor our modeling techniques to treat each in kind. We will commonly refer to these values as the *Event Variables* and the *State Variables*. A practical artifact of this approach is that we can simply use the system's existing infrastructure and logging code to generate logs, and then write a simple script to translate logs into state and event log messages in a post-processing step (§ 4.1). Adopting this approach makes it much easier to apply DISTALYZER to a wide range of existing systems, and avoids extra logging overheads at runtime. Additionally, it is possible to integrate our logging library into other logging infrastructures or code-generating toolkits that provide a distinction between state and event log messages, so that no post-processing phase would be required. Furthermore, this strategy allows the incorporation of external system activity monitoring logs for a richer analysis.

3 Design

This section presents the design of DISTALYZER, an operational tool capable of identifying salient differences between sets of logs, with the aim of focusing the attention of the developer on aspects of the system that affect overall performance and significantly contribute to the observed differences in behavior. DISTALYZER involves a multi-step analysis process as shown in Figure 1. The input to the workflow is a set of logs with tagged event and/or state log messages, separated by the developer into two classes C_0 and C_1 with different behavior on some performance metric P (*e.g.*, runtime). The choice of performance metric can be easily determined from Service Level Agreement (SLA) metrics. Some example choices for separation of classes are as follows:

- Different *versions* of the same system
- Different *requests* in the same system
- Different *implementations* of the same protocol
- Different *nodes* in the same run

DISTALYZER uses machine learning methods to automatically analyze the input data and *learn* the salient differences between the two sets of logs, as well as the relationships among the system components. Further, DISTALYZER identifies and presents to the developer (for investigation) the most notable aspects of the system likely to contain the *root cause* of the observed performance difference. Specifically the system involves the following four components:

1. **Feature Creation**: A small set of event/state *features* are extracted from each log instance (file) in both classes to make the data more amenable for automated analysis.
2. **Predictive Modeling**: The event/state variables are analyzed with statistical tests to identify which features *distinguish* the two classes of logs. This step directs attention to the system components that are the most likely causes of performance difference.
3. **Descriptive Modeling**: Within a single class of logs (*e.g.*, C_0), the relationships among event/state variables are learned with dependency networks [21].

The learned models enhance the developer's understanding of how aspects of the system interact and helps to discard less relevant characteristics (*e.g.*, background operations, randomness).

4. **Attention Focusing**: The outputs of steps 2 and 3 are combined to automatically identify a set of interrelated variables that most diverge across the logs and most affect overall performance (*i.e.*, P). The results are graphically presented to the developer for investigation, not only indicating where to look for performance bugs, but also insight into the system itself, obviating the need for the developer to be an expert at all system interactions.

We describe each of these components in more detail below. We note that the user need not be aware of the internals of the statistical or machine learning techniques, and is given an understandable graphical representation of the variables likely to contain the root cause of performance differences. With a clear understanding of the root cause, the developer can spend more time on finding a good fix for the performance bug. In Section 5 we present results of using DISTALYZER to analyze TritonSort (different versions), BigTable (different requests), and BitTorrent (different implementations).

3.1 Feature creation

The workflow starts with extracting a handful of feature summaries from the logs. The input is two sets of logs C_0 and C_1, classified by the developer according to a performance metric of interest P. For example, the developer may be interested in diagnosing the difference between slow (C_0) and fast (C_1) nodes based on total runtime P. DISTALYZER performs offline analysis on the logs, after they have been extracted from the system execution. We assume that a similar test environment was maintained for both sets of logs, including workloads, physical node setup, etc. However, it is not necessary that both classes contain the same number of occurrences of a variable. Also, the two classes are not required to have disjoint non-overlapping values for P. The necessity for similarity can be relaxed further under certain conditions (§ 7).

DISTALYZER begins by calculating features from variables extracted from the log instances. We refer to each log instance as an *instance*. Each instance i contains many event and state log messages, which first need to be summarized into a smaller set of summary statistics before analysis. The intuition behind summarizing is that this reduces the complexity of the system execution to a handful of *features* $\mathbf{X_i}$, that are much less prone to outliers, randomness and small localized discrepancies in log statements. Since DISTALYZER aims to find the source of *overall* performance problems (and not localized problems as in [37]), a coarse-grained set of features provides a better representative of each instance than every single value within that instance. A smaller set of features are a lesser burden on the developer, but a richer set provides better coverage for different types of problems. DISTALYZER aims at striking the right balance between these objectives through our experiences and intuition debugging distributed systems. DISTALYZER constructs a set of summary statistics $\mathbf{X}$ from the timestamps of event log messages and the values of numeric variables for state log messages, as described below.

Event Features The timing of system events is often closely related to overall performance, as it can identify the progress of system components, the presence or absence of noteworthy events, or the occurrence of race conditions between components. We consider a set of event variables Y^e that are recorded in the log instances with timestamps. For example, an instance may refer to a node downloading a file in BitTorrent, where the event log may contain several recv_bt_piece events over time.

To summarize the timing information associated with a particular type of event Y^e in instance i, DISTALYZER constructs features that record the time associated with the *first*, *median* and *last* occurrence in Y_i^e. (All timestamps within a log instance i are normalized based on the start time). Specifically, $X_{i.1}^e = min(\mathbf{Y_i^e}[\mathbf{t}])$, $X_{i.2}^e = median(\mathbf{Y_i^e}[\mathbf{t}])$, $X_{i.3}^e = max(\mathbf{Y_i^e}[\mathbf{t}])$. In addition, a fourth feature is constructed that counts the total number of occurrences of Y^e, $X_{i.4}^e = |\mathbf{Y_i^e}|$. Our experience debugging systems suggests that these occurrences capture some of the most useful, yet easily comprehensible, characteristics of system progress. They most commonly indicate issues including but not limited to startup delays, overall slowdown and straggling finishes.

In addition to the above features, which consider the *absolute* timing in instances, we consider the same set of features for *relative* times. Since the instances from C_0 and C_1 may have different total times, normalizing the times within each instance to the range $[0,1]$ before computing the features will yield a different perspective on event timings. For example, in BitTorrent, it is useful to know that the last outgoing connection was made at 300sec, but for debugging it may be more important to know that it occurred at 99% of the runtime when comparing to another instance where the last connection was made at 305sec, but earlier at 70% of its total runtime. In this case, the divergence in the relative event times is more distinguishing. The top half of Table 1 outlines the set of event feature types considered by DISTALYZER.

State Features It is common for some system state variables to be directly or inversely proportional to the performance, and their divergence could be equally important for diagnosis. We consider a set of state variables Y^s that maps to a list of values with their logged timestamps in an instance. For example, in BitTorrent, one of the state variables logged is the download speed of a node, which is inversely proportional to the total runtime performance. DISTALYZER does not attempt to understand the meaning behind the variables or their names, but systematically searches for patterns in the values.

To summarize the information about a particular state variable Y^s in log i, we construct features that record the *minimum*, *average* and *maximum* value in Y_i^s. Specifically, $X_{i.1}^s = min(\mathbf{Y_i^s})$, $X_{i.2}^s = mean(\mathbf{Y_i^s})$, $X_{i.3}^s = max(\mathbf{Y_i^e})$. In addition, to understand the variable values as the system progresses and also give the values context, DISTALYZER constructs features that record the variable values at one-fourth, half and three-fourth of the run. Similar to the events, the relative versions of these snapshots are also considered as feature types. The complete list of state feature types is listed in Table 1.

Event times
{First, Median, Last} × {Absolute, Relative} occurrences
{Count}
State values
{Minimum, Mean, Maximum, Final}
{One-fourth, Half, Three-fourth} × {Absolute, Relative} snapshots

Table 1: Feature types extracted from the system

Cost of Performance Differences Our analysis focuses on leveraging the characteristics of the *average* performance difference between the two classes, thus naïve use of the instances in statistical techniques will fail to distinguish performance in the *tails* of the distribution. For example, in a class of bad performance, there may be 2-3% of instances that suffer from significantly worse performance. Although these cases are relatively infrequent, the *high cost* of incurring such extreme bad performance makes analysis of these instances more important. DISTALYZER automatically detects a significant number of abnormally high/low values of the performance metric, and flags this to the developer for consideration before further analysis. Specifically, DISTALYZER identifies a "heavy" tail for P when the fraction of P_i outside $\bar{P} \pm 3\sigma_P$ is larger than 1.1% (*i.e.*, 4× the expected fraction in a normal distribution). To more explicitly consider these instances in the modeling, we can re-*weight* the instances according to a *cost* function (see e.g., [12]) that reflects the increased importance of the instances in the tail. Section 5.2.1 discusses this further.

3.2 Predictive Modeling

In the next stage of the workflow, DISTALYZER uses statistical tests to identify the features that most distinguish the two sets of logs C_0 and C_1. Specifically, for each event and state feature X described above (*e.g.*, *first*(recv_bt_piece)), we consider the distribution of feature values for the instances in each class: X_{C_0} and X_{C_1}. DISTALYZER uses t-tests to compare the two distributions and determine whether the observed differences are *significantly* different than what would be expected if the random variables were drawn from the same underlying distribution (*i.e.*, the means of X_{C_0} and X_{C_1} are equal). If the t-test rejects the null hypothesis that the $\bar{X}_{C_0} = \bar{X}_{C_1}$, then we conclude that the variable X is *predictive*, *i.e.*, able to distinguish between the two classes of interest. Specifically, we use *Welch's t-test* [36], which is defined for comparison of unpaired distributions of unequal variances. We use a critical value of $p < 0.05$ to reject the null hypothesis and assess significance, adjusting for multiple comparisons with a Bonferroni correction [11] based on the total number of features evaluated.

Our use of t-tests is motivated by the fact that we want to identify variables that distinguish the two classes on *average* across many instances from the system. Previous work [32] has used Kolmogorov-Smirnov (KS) tests to distinguish between two distributions of request flows. In that work, the bulk of the two distributions are the same and the KS test is used to determine whether there are *anomalous* values in one of the two distributions. In contrast, our work assumes that the log instances have been categorized into two distinct classes based on developer domain knowledge. Thus the overlap between distributions will be minimal if we can identify a variable that is related to performance degradation in one of the classes. In this circumstance, KS tests are too sensitive (*i.e.*, they will always reject the null hypothesis), and t-tests are more suitable form of statistical test.

Given the features that are determined to be *significant*, the magnitude of the t-statistic indicates the difference between the two distributions—a larger t-statistic can be due to a larger difference in the means and/or smaller variance in the two distributions (which implies greater separation between the two classes). The sign of the t-statistic indicates which distribution had a bigger mean. Among the significant t-tests, we return a list of significant variables ranked in descending order based on the absolute sum of t-statistic over all features. This facilitates prioritized exploration on the variables that best differentiate the two classes.

3.3 Descriptive Modeling

In the third component of the workflow, DISTALYZER learns the relationships among feature values for each class of logs separately. The goal of this component is to identify salient dependencies among the variables within a single class (*i.e.*, C_0)—to help the developer understand the relationships among aspects of the system for diagnosis and debugging, and to highlight the impact of divergent variables on overall performance P. It is often difficult to manually discover these relationships from the code, because of large code bases. It is also possible that observed variation across the classes for a feature is not necessarily related to performance. For example, a timer period may have changed between the classes without affecting the performance, and such a change can be quickly ignored if the dependencies are understood.

Since we are interested in the overall associations between the features in one class, we move beyond pairwise correlations and instead estimate the *joint distribution* among the set of features variables. Specifically, we use dependency networks (DNs) [21] to automatically learn the joint distribution among the summary statistics $\mathbf{X}$ and the performance variable P. This is useful to understand which sets of variables are inter-related based on the feature values. We construct DNs for the event and state features separately, and within each we construct two DNs for each feature type (*e.g.*, *First.Absolute*), one for instances of class C_0 and one for instances of C_1.

DNs [21] are a graphical model that represents a joint distribution over a set of variables. Consider the set of variables $\mathbf{X} = (X_1, ..., X_n)$ over which we would like to model the joint distribution $p(\mathbf{X}) = p(X_1, ..., X_n)$. Dependencies among variables are represented with a directed graph $G = (V, E)$ and conditional independence is interpreted using graph separation. Dependencies are quantified with a set of conditional probability distributions $\mathscr{P}$. Each node $v_i \in V$ corresponds to an $X_i \in \mathbf{X}$ and is associated with a probability distribution conditioned on the other variables, $p(x_i|\mathbf{x} - \{x_i\})$. The parents of node i are the set of variables that render X_i conditionally independent of the other variables ($p(x_i|pa_i) = p(x_i|\mathbf{x} - \{x_i\})$), and G contains a directed edge from each parent node v_j to each child node v_i ($(v_j, v_i) \in E$ iff $X_j \in pa_i$).

Both the structure and parameters of DNs are determined through learning the local CPDs. The DN learning algorithm learns a CPD for each variable X_i, conditioned on the other variables in the data (*i.e.*, $\mathbf{X} - \{X_i\}$). Any conditional learner can be used for this task (*e.g.*, logistic regression, decision trees). The CPD is included in the model as $\mathscr{P}(v_i)$ and the variables selected by the conditional learner form the parents of X_i (*e.g.*, if $p(x_i|\{\mathbf{x} - x_i\}) = \alpha x_j + \beta x_k$ then $PA_i = \{x_j, x_k\}$). If the conditional learner is not selective (*i.e.*, the algorithm does not select a subset of the features), the DN will be fully connected. To build understandable DNs, it is thus desirable to use a selective learner. Since event and state features have continuous values, we use Regression Trees [10] as the conditional learner for the DNs, which have an advantage over standard regression models in that they are selective models.

Improvements The graphical visualization of the learned DN are enhanced to highlight to the developer (1) the divergence across classes (sizes of the nodes), (2) the strength of associations among features (thickness of edges), and (3) temporal dependencies among features (direction of edges). Specifically, each feature (node) in the DN is matched with its corresponding statistical t-test value. Since the t-statistics reflect the amount of divergence in the feature, across the two classes of logs, they are used to size the nodes of the graph. Next, for the assessment of relationship strength, we use an input parameter m for the regression tree that controls the minimum number of training samples required to split a leaf node in the tree and continue growing (*i.e.*, a large value of m leads to *shorter* trees because tree growth is stopped prematurely). The dependencies identified in a shorter tree are *stronger* because such variables are most correlated with the target variable and affect a larger number of instances. Thus, we weigh each edge by the value of m for which the relationship is still included in the DN. Finally, we augment the DN graphical representation to include *happens-before* relationships among the features. If a feature value X_i occurs before feature value X_j in all log instances, the edge between X_i and X_j is drawn as directed in the DN.

3.4 Attention Focusing

The final component of the workflow automatically identifies the most notable results to present to the user. The goal of this component is to focus the developers attention on the most likely causes of the observed performance differences. The predictive modeling component identifies and presents a ranked list of features that show significant divergences between the two classes of logs. The divergence of a single feature is usually not enough to understand both the root cause of performance problems and their impact on performance—because performance problems often manifest as a causal chain, much like the domino effect. The root cause feature initiates the divergence and forces associated features (down the causal chain) to diverge as well, eventually leading to overall performance degradation.

Moreover, we noticed that divergences tend to increase along a chain of interrelated features, thus the root cause may not have the largest divergence (*i.e.*, it may not appear at the top of the ranking). The descriptive modeling component, on the other hand, identifies the associations among features within a single class of logs. These dependencies can highlight the features that are associated with the performance measure P. To identify likely *causes* for the performance difference, DISTALYZER searches for a small set of features that are *both* highly divergent and have strong dependencies with P. The search procedure for finding the DN that highlights this set is detailed below.

The set of DNs vary across three dimensions: (1) event *vs.* state features, (2) feature type, *e.g.*, *First.Absolute*, and (3) the parameter value m_{min} used to learn the DN. In our experiments, we set m_{min} to one-third of the instances. The aim was to focus on the sufficiently strong relationships among features, and this choice of m_{min} consistently proved effective in all our case studies. However, m_{min} is included as a tunable parameter in the system for the developer to vary and observe the impact on the learned models. DISTALYZER identifies the most notable DN graph for the state and event features separately. Within a particular set, the attention-focusing algorithm automatically selects the feature type with the "best" scoring DN subgraph. To score the DN graphs, they are first pruned for the smallest connected component containing the node P, and then the selected components are scored using Algorithm 1.

The intuition behind the DN subgraph *score* function is that it should increase proportionally with both the node weights (divergence across classes) and the edge weights (strength of association). The node and edge weights are normalized before computing this score. If the developer is interested in biasing the search toward features with larger divergences or toward stronger dependencies, a parameter α can be used to moderate their relative contributions in the score. The feature type with the highest scoring connected component is selected and returned to the developer for inspection.

Algorithm 1 Feature Scoring for Dependency Networks

```
Input: Log type: $t$ (State / Event)
Input: Log class: $c$, Number of instances: $N$
Input: T-tests for all random variables in $(t, c)$
Input: DNs for all random variables in $(t, c)$
Input: Performance metric: $P$
  $feature\_graphs = \{\}$
  for Feature $f$: feature_types($t$) do
    $dn$ = $\mathsf{DN}_f(m_{min} = N/3)$
    $cc$ = Connected-component in $dn$ containing $P$
    $tree$ = maxSpanningTree($cc$) rooted at $P$
    $score = 0$
    for Node $n$: $tree$ do
      $score$ += $\mathsf{T}_f(n) * dn$.weight(parentEdge($n$))
    end for
    Append $(score, cc)$ to $feature\_graphs$
  end for
  return $feature\_graphs$ sorted by score
```

Section 5 describes the outputs of DISTALYZER for real systems with observed performance problems. Apart from the final output of the attention focusing algorithm, the developer can also access a table of all the t-test values and dependency graphs for both the state and event logs. This is shown as the final stage in Fig. 1.

```
setInstance(class, instance_id)
logStateValue(timestamp, name, value)
logEventTime(timestamp, name)
```

Figure 2: DISTALYZER logging API

4 Implementation

We describe some implementation details for transforming text logs and developing DISTALYZER.

4.1 Processing Text Log messages

The BitTorrent implementations we considered were implemented in C (Transmission) and Java (Azureus), whereas HBase was implemented in Java. The Java implementations used Log4j as their logger. Transmission however used hand-coded log statements. HBase also used Log4j, but did not have any logs in the request path.

For each implementation, we tailored a simple Perl script to translate the text logs into a standard format that DISTALYZER accepts. We maintained a simple internal format for DISTALYZER. This format captures the timestamp, type of log, and the name of the log. For state logs, the format additionally includes the value of the log. We advocate adopting a similar procedure for analyzing any new system implementation. A developer with domain knowledge on the system should be able to write simple one-time text parsers to translate the most important components of the log instances. To support the translation, we provide a simple library API for logging in a format accepted by DISTALYZER (shown in Fig. 2). At the beginning of each log instance, the translator calls setInstance, which indicates the instance id and class label for subsequent log messages. It specifically requires marking log messages as event or state logs at translation time by calling one of the two log methods.

4.2 DISTALYZER

We implemented DISTALYZER in Python and C++ (4000 lines of code) using the scientific computing libraries Numpy and Scipy. It is publicly available for download [1]. The design allows adding or tweaking any of the event or state features if required by the developer. The Orange data mining library [10] provides regression tree construction, and we implemented dependency networks and Algorithm 1 over that functionality. The DOT language is used to represent the graphs, and Graphviz generates their visualizations. The implementation of DISTALYZER comprises of many embarrassingly parallel sub-tasks and can easily scale on multiple cores and machines enabling quick processing.

An interactive JavaScript based HTML interface is presented to the developer along with the final output. This immensely helps in trudging through the individual distributions of variables, and also to view the dependency graphs of all features. This has been useful in the post-root cause debugging process of finding a possible fix for the issue. To a good extent, this also helps in understanding some of the non-performance related behavioral differences between the logs. For example, in one case of comparing different implementations, we noticed that either system was preferring the use of different protocol messages to achieve similar goals.

5 Case Studies

Our goal in these case studies is to demonstrate that DISTALYZER can be applied simply and effectively to a broad range of existing systems, and that it simplifies the otherwise complex process of diagnosing the root cause of significant performance problems. We therefore applied DISTALYZER across three real, mature and popular distributed systems implementations. Table 2 captures the overview of the systems we considered. These systems represent different types of distributed system applications: distributed sorting, databases, and file transfers. We identified previously unknown performance problems with two of these systems, and worked with an external developer to evaluate usefulness of DISTALYZER in rediscovering a known performance bug with another. In all cases, DISTALYZER significantly narrowed down the space of possibilities without the developer having to understand all components. Due to space constraints, we are unable to describe each action taken by the developer leading to fixes for problems. A user need not be aware of how the tool computes divergences and dependencies to understand DISTALYZER's outputs. We describe the outputs of DISTALYZER and henceforth straightforward debugging process.

5.1 TritonSort

TritonSort is a large scale distributed sorting system [30] designed to sort up to 100TB of data, and holds four 2011 world records for 100TB sorting. We demonstrate the effectiveness of DISTALYZER by applying it over logs from a known bug. We obtained the logs of TritonSort from the authors, taken from a run that suddenly exhibited 74% slower performance on a day. After systematically and painstakingly exploring all stages of the sort pipeline and running micro-benchmarks to verify experimental scenarios, the authors finally fixed the problem. They said that it took "the better part of two days to diagnose". The debugging process for the same bug took about 3-4hrs using DISTALYZER, which includes the implementation time of a log parser in 100 lines of Python code. A detailed analysis of the output of DISTALYZER and the debugging process on these logs follows.

We had access to logs from a 34 node experiment from the slow run that took 383 sec, and also a separate run with the same workload that had a smaller runtime of 220 sec. These naturally fit into two classes of logs with one instance per node, which could be compared to identify the reason for the slowdown. These logs were collected as a part of normal daily testing, meaning no additional overhead for log collection. The logs contained both event and state log messages that represented 8 different stages of the system (Table 2). The performance metrics were identified as Finish and runtime for the event and state logs respectively, both indicating the time to completion. Fig. 3 shows the final dependency sub-graphs output by DISTALYZER for both event and state logs.

To briefly explain the visualization generated by DISTALYZER, nodes shown to be colored indicate the performance metric and the font size is proportional to the magnitude of the divergence. Edge thickness represents the strength of the dependencies between variables. Directed edges in event graphs indicate that a *happens-before* relationship was identified between the two bounding variables, as described in Section 3.4.

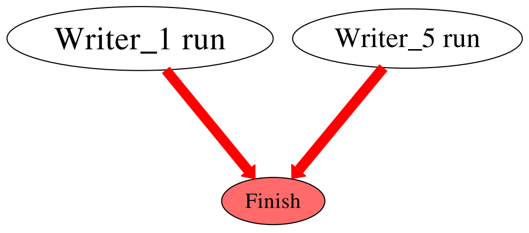
The best dependency graph picked for events (*Last* feature type) is shown in Fig. 3a, indicating that variables Writer_1 run and Writer_5 run are both significant causes of Finish's divergence. The final stage of TritonSort's pipeline is the writer which basically handles writing the sorted data to the disk. Each stage in TritonSort is executed by multiple thread workers, denoted by the number in the variable. This analysis attributes the root cause of slow runs to highly divergent last occurrences of the writer workers. A quick look at our distribution comparison of the two sets of logs in both the writers indicated that the slow run showed a difference of 90 sec. The performance metric and the writer run distributions also showed an outlier with a larger time than the rest.

Similarly, the DN picked for the states is shown in Fig. 3b, where the performance metric Runtime is connected to the subgraph consisting of the write queue size of different writer workers. Although the figure was scaled down for space constraints, it is clear that all the nodes are highly divergent like the total performance. To understand the reason of this divergence, we looked at distributions for Absolute Half (best feature) to learn that writers in the slow run were writing 83% more data. Thus, we concluded the root cause as *slow writers*.

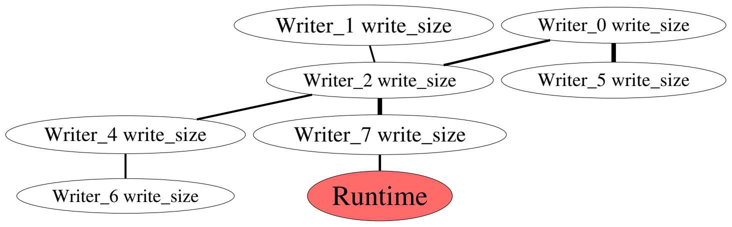
The actual bug had been narrowed down to the disk writing stage, found to be slowing down earlier stages of the pipeline. It was further noticed that a single node was causing most of this delay, which eventually led the authors to discover that the cache battery on that node had disconnected. This resulted in the disks defaulting to write-through and hence the poor performance. Both the top ranked DNs output by DISTALYZER were useful in identifying the bug. We shared these DNs and interactive t-test tables with the author of the paper, who had

System Implementation	Types of Logs	Volume	Variables	Issues	Performance gain	New issues
TritonSort	State, Event	2.4 GB	227	1	n/a	×
HBase (BigTable)	Event	2.5 GB	10	3	22%	√
Transmission (BitTorrent)	State, Event	5.6 GB	40	2	45%	√

Table 2: Summary of performance issues identified by DISTALYZER



(a) Event



(b) State

Figure 3: TritonSort dependency graphs indicating the root cause of the slow runtime

manually debugged this problem. The output root cause was immediately clear to him, and he surmised "had we had this tool when we encountered this problem, it would have been a lot easier to isolate the difference between the bad run and a prior good one".

5.2 HBase

BigTable [5] is a large-scale storage system developed by Google, holds structured data based on rows and columns, and can scale efficiently to a very large number of rows and column content. HBase [20] is an open source implementation of BigTable being developed by the Apache foundation. It runs on top of Hadoop Distributed Filesystem (HDFS), and has been tuned and tested for large scales and performance.

In our experiments, we noticed that "Workload D" from the Yahoo Cloud Storage Benchmark (YCSB) [9] had a notable heavy tail distribution of read request latencies. The minimum and median latencies were 0 and 2 msec respectively. However the mean latency was 5.25 msec and the highest latency was as high as 1 second, which is 3 orders of magnitude over the median. Moreover, more than 1000 requests have a latency greater than 100ms. To debug this performance bottleneck in HBase, we would like to be able to compare these slow requests to the huge bulk of fast ones. This task is infeasible manually because these issues manifest only in large experiments (1 million requests), and a sufficiently large number of requests exhibit this behavior. We used DISTALYZER to identify and debug three performance bugs in HBase, two of which are described below in detail.

Experimental setup Our testbed consisted of 10 machines with 2.33GHz Intel Xeon CPUs, 8GB RAM and 1Gbps Ethernet connections running Linux 2.6.35.11. Our HBase setup used a single master on a dedicated machine, and 9 region servers (equivalent to BigTable tablet servers), and 1 Million rows of 30kB each were pre-loaded into the database. The YCSB client was run on the same machine as the master (which was otherwise lightly loaded), with 10 threads issuing parallel requests. Each request is either a read or write for a single row across all columns. "Workload D" consisted of 1 Million operations out of which 5% were writes.

The HBase implementation had no log statements in the request flow path, in spite of using the *log4j* logging library that supports log levels. Therefore, we manually added 10 event logs to the read request path, using the request row key as the identifier. The request logs from the different machines were gathered at the end of the run and bucketed by request ID. The performance metric is the event that signifies the last step in request processing – HBaseClient.post_get.

5.2.1 Fixing the slowest outliers

On applying DISTALYZER to the logs, it detected the presence of a heavy tail in the performance metric (§ 3.1) and suggested re-weighting the instances. The weight function used to boost the instances with a large latency was $\lfloor w^{latency} \rfloor$. This is an exponential weight function and we chose a value of $w = 2^{(1/150)}$, with the intuition that instances with $P < 150ms$ will have a weight of 1. Fig. 4 shows the best DN of the root cause divergence. All dependency edges are directed because all requests follow the same flow path through the system. We identified two strong associations with large divergences leading up to the performance metric. Each of the chains is considered independently, and we first chose to follow the path leading from client.HTable.get_lookup (the second chain is discussed in § 5.2.2). This chain starts at client.HTable.get which indicates that the HBase client library received the request from YCSB, followed by client.HTable.get_lookup after completion of lookup for the region server handling the given key.

This particular edge leads from a tiny variable to a variable with significant divergence, and domain knowl-

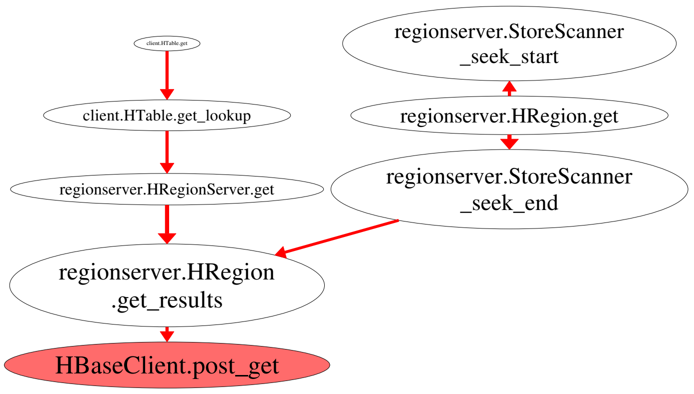


Figure 4: DN for unmodified HBase events

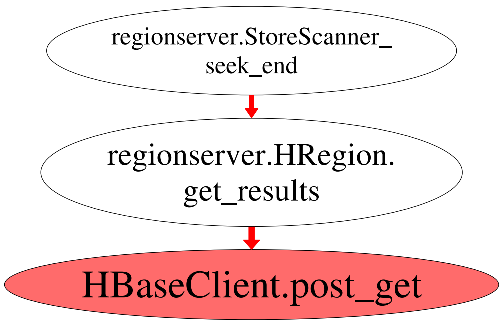


Figure 5: DN for HBase after fixing lookups

edge indicates that no other event occur between them. client.HTable.get is drawn small because it does not differ considerably between the two classes of logs. As it is connected by a strong directed edge to the larger variable, this indicates the two classes consistently *differ* between these two variables. In this context, the edge represents the operation where the client needs to lookup the particular region server that manages the row, and this is achieved by contacting the master who maintains the mapping. The distributions of this particular event in the t-test table shows that this event created gaps in the request flow of the order of 1000 ms.

When we looked at the logs of the regionserver at the same time these requests were being delayed, we noticed that the server was throwing a NotServingRegionException. This is given by the server when it does not serve a region that was specifically requested. This happens when a region was moved to another server for load balancing. The client possesses a stale cache entry for the region, and hence receives this exception. The client was catching this exception as an IOException, and treated it as a server failure. This triggers an exponential back off procedure that starts at 1 sec. According to the Bigtable description [5], the client immediately recognizes a stale cache and retries with the master leading to an overhead of just 2RTTs. We came up with a fix for this issue, by treating the exceptions correctly and extracting the NotServingRegionException, and retrying immediately. This fixed the requests with latencies over 1 second.

5.2.2 Operating System effects

DISTALYZER was used again to analyze the new logs to find the cause of the other delays. Since the distribution skew was lesser than the threshold, the weighting function was not used anymore. The best DN is shown in Fig. 5, and closely resembles the right chain of Fig. 4. In fact, this root cause was also identified in the initial step as a second significant root cause, but was not chosen for inspection. Here, the variables regionserver.StoreScanner_seek_end and regionserver.HRegion.get_results chain up as the root cause.

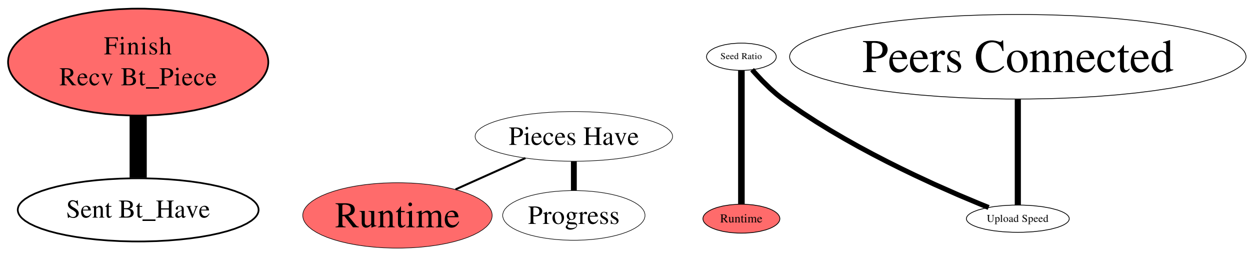
The default Linux I/O scheduler since version 2.6.18 is Completely Fair Queuing (CFQ), and it attempts to provide fairness between disk accesses from multiple processes. It also batches requests to the disk controller based on the priority, but it does not guarantee any completion times on disk requests. Since only the HBase process was accessing the disk on these machines, we believed that this scheduling policy was not well suited to random block reads requested by HBase. Another available I/O scheduler in Linux is the deadline scheduler, which tries to guarantee a start service time for requests. Hence the deadline scheduler would be more suited toward latency sensitive operations.

After we applied the I/O scheduler change, we ran the same experiment again to understand if this improved the latencies of the slow requests. The number of slow requests ($\geq$100ms) reduced from 1200 to just under 500 – a 60% reduction. Also, the mean latency for the workload dropped from 5.3ms to 4ms, which is a 25% overall improvement in the read latency, confirming deadline is appropriate for these workloads. Both the reported root cause DNs were helpful in debugging HBase.

Further, we identified a problem with HBase's TCP networking code which affected latencies of requests, but we do not discuss it here for brevity.

5.3 Transmission

Transmission implements the BitTorrent protocol, a distributed file sharing mechanism that downloads different pieces of a file from multiple peers. The protocol works by requesting a set of active peers for the file from a *tracker*, then directly requests file pieces for download from them. By downloading from multiple peers simultaneously, clients can more easily download at large speeds limited only by its bandwidth. Azureus is another BitTorrent implementation, that we used for comparison. In some basic experiments, Transmission had a much worse download time compared to Azureus (552 sec vs. 288 sec).



(a) Event DN *(b)* State DN: 1st with score 0.89 *(c)* State DN: 2nd with score 0.84

Figure 6: Dependency graphs for unmodified Transmission

Transmission [35] is a light-weight C implementation, and among all the free clients, it is known for its minimal resource footprint. Azureus [3] is one of the most popular free implementations of the protocol, developed in Java. It is an older and more mature implementation of the protocol and well known for its excellent performance. Unlike Transmission, it extends the basic BitTorrent messaging protocol for extra minor optimizations in communicating with supporting peers. Both are serious implementations of the protocol, and we expect a well tuned C implementation should perform no worse than a Java implementation. Using DISTALYZER, we were able to identify two performance bugs in Transmission that eliminated the download time difference completely.

Experimental setup Experiments consisted of 180 BitTorrent clients (30 clients per machine) attempting to download a 50MB file, providing ample interaction complexity in the system. They used the same machines as described in Sec. 5.2. The swarm was bootstrapped with a single seeder, and each client was limited to an upload bandwidth of 250KB/s which is similar to common Internet bandwidths and makes ample room for running 30 clients on a single machine. Experiments were conducted with each implementation in isolation.

We built Azureus from its repository at rev. 25602 (v4504). Azureus had a detailed log of BitTorrent protocol messages during a download, and we added some state logs. The experiments used the HotSpot Server JVM build 1.6.0_20. We used version 2.03 of Transmission in our experiments, which contained debugging logs, and we simply activated the ones pertaining to the BitTorrent protocol. We identified the event and state performance metrics Finish and Runtime, respectively.

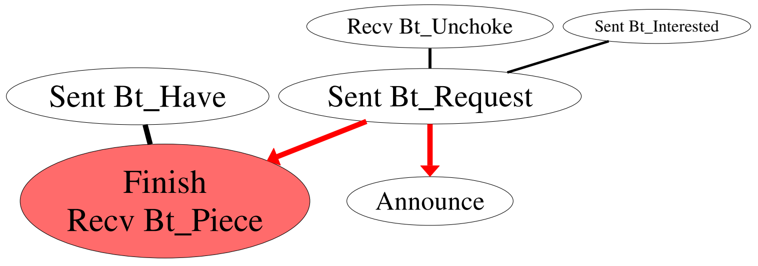
5.3.1 Faulty component affecting performance

The best DNs output by DISTALYZER for both event and state shown in Fig. 6a, 6b were dependencies between trivial divergences. These are in a sense false positives to the automatic root cause detection. More specifically, Fig. 6a was picked from the *Last* event-feature and shows the performance metric coalesced with the last piece receipt. The strong dependency to Sent_Bt_Have is justified by the fact that implementations send out piece advertisements to peers, as soon as they receive one more piece. Similarly, the state dependency graph in Fig. 6b shows strong dependencies between download completion time and the number of pieces download in half the run, and also the progress (which is in fact a factor of Pieces Have). We discard these DNs and move to lesser ranks.

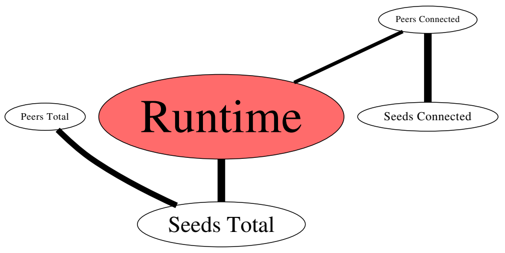
This led to considering the second ranked state graph in Fig. 6c, which in fact had a very close score to the highest rank. This DN was constructed from snapshots of the state variables at three-fourth of Transmission's runtime. Runtime is connected to divergent Peers Connected through a chain of variables. The chain involves the amount of data seeded and upload speed, both affirming the symbiotic nature of BitTorrent. This immediately takes us to the distributions of the number of peers, where we noticed that all nodes reported 6 peers in Transmission, as against 50 for Azureus. We also verified these values for the *Maximum* feature.

Fixing the bug To find the problem that limited Transmission's peer connectivity, we considered a single node's logs and fetched the set of unique IP:port pairs, and on looking at the values, we immediately realized that each peer had a different IP address. In our experimental setup with 6 physical machines, different nodes on the same physical machine were setup to listen on different ports and coexist peacefully. The bug was traced to the internal set that holds peers, whose comparison function completely ignored port numbers. When a node obtains a new peer from the tracker, and it is already connected to a peer with the same IP address, it is simply dropped.

On looking through forums and bug management softwares, we found that this inconsistency had actually been identified 13 months back, but the bug was incorrectly closed. We verified the authenticity of this bug and reopened it. The developers deemed this bug to be hard to fix, in terms of requiring changes to many modules. We argue that this is an important bug that limits Transmission from connecting to multiple peers behind a NAT



(a) Event



(b) State

Figure 7: Dependency graphs for BitTorrent after fixing the NAT problem

box. In cases where multiple peers are situated behind a NAT box in an ISP, they would definitely want to download from each other and avoid the slow ISP link. This bug would prevent local connections, thus forcing them to connect to peers on the Internet.

5.3.2 Tuning the performance

Since the fix for the first bug was too tedious, we decided to circumvent the problem by assigning unique virtual IP addresses to each of the nodes. This did indeed solve the problem and made Transmission faster to an average download time of 342 sec, which was still much higher than 288 sec. DISTALYZER was used again with the new set of logs which produced the dependency graph output shown in Fig. 7. Considering the event DN in Fig. 7a, showing the highly divergent performance metric for the *Last* feature. Some of the features of this DN are similar to Fig. 6a that were discussed earlier.

The dependency between finishing and sending requests fits well with the protocol specifications, that a request for a piece must be sent in order to receive one. The Announce event happens *after* sending out requests, and hence de-values its possibility for root cause. The interested messages were a more probable cause of the differences (compared to un-choke) because one must first express interest in another peer after connection establishment. Only after this step does the remote peer un-choke it, thus opening up the connection to piece requests. This hypothesis was verified by viewing the distributions of Sent_Bt_Interested across all features. After knowing the root cause, the distribution for the offending variable in the *First* feature showed gaps of the order of 10 sec on Transmission, but was very small for Azureus.

We traced the code from the message generator to fix these large gaps, and found a timer (called rechokeTimer) that fired every 10 sec. For comparison, we found that Azureus had a similar timer set at 1 sec, thus giving it a quicker download start. The large divergence in sending interested messages could be fixed by shortening the timer value from 10sec to 1sec. Fig. 7b shows the state DN for the same logs for completeness, but it does not indicate a highly divergent root cause.

Performance gains We were able to apply a quick fix for this problem and the download times of Transmission were much better than earlier, dropping the mean completion time to 288 sec. The performance was up to 45% better than the first experiment. It should be noted that the more frequent timer did not affect the resource utilization of Transmission, still using far fewer CPU cycles and memory than Azureus. Neither of these issues affected correctness, nor threw any sort of exceptions, and present themselves as subtle challenges to the developers. Overall, 5 DNs were reported for the two issues in Transmission, out of which 3 indicated trivial relationships between the components, but the other two were immensely helpful in understanding the root causes.

6 Related Work

Model checking aims to provide guarantees on program code against pre-specified properties. A number of techniques [18, 23, 28] have described different methods to assert program correctness. However, traditional model checking attempts to discover violations of clear failure conditions. There is also research in applying machine learning to logs of faulty executions, to categorize them [4,8] and also predict the root cause [6]. Conditions of performance degradation cannot be accurately modeled using these approaches, because it is rarely possible to specify performance as definite runtime predicates.

The formulation of debugging as an anomaly detection task has been applied in a variety of contexts. Magpie [4] and Pinpoint [6] model request paths in the system to cluster performance behaviors, and identify root causes of failures and anomalous performance. Fu *et al.* [14] propose the use of a Finite State Automaton to learn the structure of a normal execution, and use it to detect anomalies in performance of new input log files. Xu *et al.* [37] propose a mechanism to encode logs into state ratio vectors and message count vectors, and apply Principal Component Analysis to identify anomalous patterns within an execution. However, they completely ignore timestamps in logs and use the value logged, to identify localized problems within a single log file. On the other hand, DISTALYZER finds the root cause of the most significant performance problem that affects the *overall* performance. In contrast to all these systems, DISTALYZER aims to find the cause of performance problems in a major portion of the log instances, and hence uses t-tests to compare the average performance.

Request flows are a specific type of distributed processing, with a pre-defined set of execution path *events* in the system. Sambasivan *et al.* [32] aim to find structural and performance anomalies in request flows that are induced by code changes. Their approach of comparing different requests bears some similarity to our technique. However, as we illustrate through our case studies, DISTALYZER can be applied to request flow systems (HBase), as well as other types of distributed systems, by abstracting the logs into states and events. Although these specific applications of machine learning (including [2,4,6,7]) can leverage path structures, DISTALYZER can show the *most impacting* root cause among many performance problems.

Cohen *et al.* [7] use instrumentation data from servers to correlate bad performance and resource usage using tree-augmented Bayesian networks. Similarly, DISTALYZER can utilize system monitoring data as outlined in Section 2 to identify performance slowdowns due to resource contention using DNs. NetMedic [22] and Giza [27] use machine learning to construct dependency graphs of networked components, to diagnose faults and performance problems. WISE [34] uses network packet statistics to predict changes to CDN response times on configuration changes, using causal Bayesian networks. In contrast, the use of distributed system logs allows DISTALYZER to identify software bugs by marking specific components in the code. Our novel use of dependency networks to learn associations between code components alleviates the need for an expert developer.

Splunk [33] is an enterprise software for monitoring and analyzing system logs, with an impressive feature set. Although it provides a good visual interface for manually scanning through logs and finding patterns, it does not provide tools for rich statistical analysis on the data. Furthermore, there is no support for comparing two sets of logs automatically. We believe that Splunk is complementary to our work, and the concepts embodied in DISTALYZER could serve as a great addition to Splunk.

7 Practical Implications

While DISTALYZER has proven to be useful at finding issues in real systems implementations, we now discuss some of the practical implications of our approach, to illustrate when it is a good fit for use.

First, DISTALYZER is based on comparing many log instances using statistical approaches. To be effective, there must exist enough samples of a particular behavior for the tool to determine that a behavior is not just a statistical anomaly. The use of weights is a partial solution to this problem. Similarly, however, the tool cannot find problems which are not exercised by the logs at all, either originating from an external black box component or insufficient logging within the system. In the former case, there is hope that existing logs would capture artifacts of the external problem and hence point to that component. The ideal approach would be combining logs from the external component or network with the existing logs, to paint the complete picture. With insufficient logging, DISTALYZER would fail to find feature(s) that describe the performance problem. This can be alleviated with additional instrumentation followed by iterative use of DISTALYZER to diagnose the issue.

Second, we assume similar execution environments for generating the logs, leaving situations of differing machine architectures, network setups or node count in obscurity. This is a tricky process because a subset of features can be dependent on the environment, and hence their divergence would be trivial leading to futile DNs. As a counter measure, these features can either be removed or transformed into a comparable form with domain knowledge. The specific case of *relative* times for event features highlights such a transformation. In future work, we imagine support for a mapping technique provided by the user for converting the features into comparable forms, allowing DISTALYZER to be used even to compare different environments.

Finally, the system inherently requires log data. If it is impractical to collect logs, either due to the overhead imposed or the manual effort required to instrument un-instrumented systems, our tool will not be a good choice. Similarly, it is important when using DISTALYZER to verify that the user-provided classifying distribution is not adversely affected by the instrumentation. Indeed, one “problem” we tracked down using DISTALYZER identified that some poor performance was actually caused by the system’s logging infrastructure flushing to disk after every log call. This is observed by seeing performance variations with and without logging.

8 Conclusion

This paper proposes a technique for comparing distributed systems logs with the aim of diagnosing performance problems. By abstracting simple structure from the logs, our machine learning techniques can analyze the behavior of poorly performing logs as divergence from a given baseline. We design and implement DISTALYZER, which can consume log files from multiple nodes, implementations, runs and requests and visually output the most significant root cause of the performance variation. Our analysis of three mature and popular distributed systems demonstrates the generality, utility, and significance of the tool, and the reality that even mature systems can have undiagnosed performance issues that impact the overhead, cost, or health of our systems. DISTALYZER can help to find and solve these problems when manual analysis is unsuccessful.

9 Acknowledgments

We would like to thank Alex Rasmussen for assisting us with using DISTALYZER on TritonSort logs, and our anonymous reviewers for their feedback and comments. This research was supported in part by the National Science Foundation (NSF) under grant CNS-1054567. Any opinions, findings and conclusions or recommendations in this paper are those of the authors and do not necessarily reflect the views of the NSF.

References

[1] Distalyzer download. http://www.macesystems.org/distalyzer/.

[2] AGUILERA, M. K., MOGUL, J. C., WIENER, J. L., REYNOLDS, P., AND MUTHITACHAROEN, A. Performance debugging for distributed systems of black boxes. In *SOSP* (2003).

[3] Azureus BitTorrent Client. http://azureus.sourceforge.net/.

[4] BARHAM, P., DONNELLY, A., ISAACS, R., AND MORTIER, R. Using Magpie for Request Extraction and Workload Modelling. In *Proceedings of OSDI* (2004).

[5] CHANG, F., DEAN, J., GHEMAWAT, S., HSIEH, W. C., WALLACH, D. A., BURROWS, M., CHANDRA, T., FIKES, A., AND GRUBER, R. E. Bigtable: A Distributed Storage System for Structured Data. In *Proceedings of OSDI* (2006).

[6] CHEN, M. Y., ACCARDI, A., KICIMAN, E., LLOYD, J., PATTERSON, D., FOX, A., AND BREWER, E. Path-Based Failure and Evolution Management. In *Proceedings of NSDI* (2004).

[7] COHEN, I., GOLDSZMIDT, M., KELLY, T., SYMONS, J., AND CHASE, J. S. Correlating Instrumentation Data to System States: A Building Block for Automated Diagnosis and Control. In *Proceedings of OSDI* (2004), USENIX Association, pp. 16–16.

[8] COHEN, I., ZHANG, S., GOLDSZMIDT, M., SYMONS, J., KELLY, T., AND FOX, A. Capturing, indexing, clustering, and retrieving system history. In *SOSP* (2005), ACM, pp. 105–118.

[9] COOPER, B. F., SILBERSTEIN, A., TAM, E., RAMAKRISHNAN, R., AND SEARS, R. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of SOCC* (2010).

[10] DEMŠAR, J., ZUPAN, B., LEBAN, G., AND CURK, T. Orange: From Experimental Machine Learning to Interactive Data Mining. In *Proceedings of PKDD*. 2004.

[11] DUNN, O. J. Multiple Comparisons Among Means. *Journal of the American Statistical Association 56*, 293 (1961), 52–64.

[12] ELKAN, C. The Foundations of Cost-Sensitive Learning. In *IJCAI* (2001), pp. 973–978.

[13] FONSECA, R., PORTER, G., KATZ, R. H., SHENKER, S., AND STOICA, I. X-trace: A Pervasive Network Tracing Framework. In *Proceedings of NSDI* (2007), USENIX Association.

[14] FU, Q., LOU, J.-G., WANG, Y., AND LI, J. Execution Anomaly Detection in Distributed Systems through Unstructured Log Analysis. In *Proceedings of ICDM* (2009), pp. 149 –158.

[15] GEELS, D., ALTEKAR, G., MANIATIS, P., ROSCOE, T., AND STOICA, I. Friday: Global Comprehension For Distributed Replay. In *Proceedings of NSDI* (2007).

[16] GEELS, D., ALTEKAR, G., SHENKER, S., AND STOICA, I. Replay Debugging for Distributed Applications. In *Proceedings of Usenix ATC* (2006).

[17] GHEMAWAT, S., GOBIOFF, H., AND LEUNG, S.-T. The Google File System. *SIGOPS Oper. Syst. Rev. 37*, 5 (2003), 29–43.

[18] GODEFROID, P. Model Checking for Programming Languages using VeriSoft. In *Proceedings of POPL* (1997).

[19] Apache Hadoop Project. http://hadoop.apache.org/.

[20] Hbase. http://hbase.apache.org/.

[21] HECKERMAN, D., CHICKERING, D. M., MEEK, C., ROUNTHWAITE, R., AND KADIE, C. Dependency Networks For Inference, Collaborative Filtering, and Data Visualization. *JMLR* (2001), 49–75.

[22] KANDULA, S., MAHAJAN, R., VERKAIK, P., AGARWAL, S., PADHYE, J., AND BAHL, P. Detailed Diagnosis in Enterprise Networks. In *Proceedings of ACM SIGCOMM* (2009), pp. 243–254.

[23] KILLIAN, C., ANDERSON, J. W., JHALA, R., AND VAHDAT, A. Life, Death, and the Critical Transition: Detecting Liveness Bugs in Systems Code. In *Proceedings of NSDI* (2007).

[24] KILLIAN, C., NAGARAJ, K., PERVEZ, S., BRAUD, R., ANDERSON, J. W., AND JHALA, R. Finding Latent Performance Bugs in Systems Implementations. In *Proc. of FSE* (2010).

[25] LIU, X., GUO, Z., WANG, X., CHEN, F., LIAN, X., TANG, J., WU, M., KAASHOEK, M. F., AND ZHANG, Z. D^{3}S: Debugging Deployed Distributed Systems. In *Proc. of NSDI* (2008).

[26] Apache log4j. http://logging.apache.org/log4j.

[27] MAHIMKAR, A. A., GE, Z., SHAIKH, A., WANG, J., YATES, J., ZHANG, Y., AND ZHAO, Q. Towards Automated Performance Diagnosis in a Large IPTV Network. In *Proceedings of ACM SIGCOMM* (2009), pp. 231–242.

[28] MUSUVATHI, M., PARK, D. Y. W., CHOU, A., ENGLER, D. R., AND DILL, D. L. CMC: A Pragmatic Approach to Model Checking Real Code. In *Proceedings of SOSP* (2002), ACM.

[29] MUSUVATHI, M., QADEER, S., BALL, T., BASLER, G., NAINAR, P. A., AND NEAMTIU, I. Finding and Reproducing Heisenbugs in Concurrent Programs. In *Proc. of OSDI* (2008).

[30] RASMUSSEN, A., PORTER, G., CONLEY, M., MADHYASTHA, H. V., MYSORE, R. N., PUCHER, A., AND VAHDAT, A. TritonSort: A Balanced Large-Scale Sorting System. In *NSDI* (2011).

[31] REYNOLDS, P., KILLIAN, C., WIENER, J. L., MOGUL, J. C., SHAH, M. A., AND VAHDAT, A. Pip: Detecting The Unexpected In Distributed Systems. In *Proceedings of NSDI* (2006).

[32] SAMBASIVAN, R. R., ZHENG, A. X., ROSA, M. D., KREVAT, E., WHITMAN, S., STROUCKEN, M., WANG, W., XU, L., AND GANGER, G. R. Diagnosing Performance Changes by Comparing Request Flows. In *Proceedings of NSDI* (2011).

[33] Splunk. http://www.splunk.com/.

[34] TARIQ, M., ZEITOUN, A., VALANCIUS, V., FEAMSTER, N., AND AMMAR, M. Answering What-If Deployment and Configuration Questions with WISE. In *Proceedings of ACM SIGCOMM* (2008), pp. 99–110.

[35] Transmission BitTorrent Client. http://www.transmissionbt.com/.

[36] WELCH, B. L. The Generalization of Student's Problem when Several Different Population Variances are Involved. *Biometrika 34*, 1-2 (1947), 28–35.

[37] XU, W., HUANG, L., FOX, A., PATTERSON, D., AND JORDAN, M. I. Detecting Large-Scale System Problems by Mining Console Logs. In *Proceedings of SOSP* (2009), ACM, pp. 117–132.

[38] YABANDEH, M., KNEZEVIC, N., KOSTIC, D., AND KUNCAK, V. CrystalBall: Predicting and Preventing Inconsistencies in Deployed Distributed Systems. In *Proceedings of NSDI* (2009).

Orchestrating the Deployment of Computations in the Cloud with Conductor

Alexander Wieder *Pramod Bhatotia* *Ansley Post*[1] *Rodrigo Rodrigues*[2]
Max Planck Institute for Software Systems (MPI-SWS)

Abstract

When organizations move computation to the cloud, they must choose from a myriad of cloud services that can be used to outsource these jobs. The impact of this choice on price and performance is unclear, even for technical users. To further complicate this choice, factors like price fluctuations due to spot markets, or the cost of recovering from faults must also be factored in. In this paper, we present Conductor, a system that frees cloud customers from the burden of deciding which services to use when deploying MapReduce computations in the cloud. With Conductor, customers only specify goals, e.g., minimizing monetary cost or completion time, and the system automatically selects the best cloud services to use, deploys the computation according to that selection, and adapts to changing conditions at deployment time. The design of Conductor includes several novel features, such as a system to manage the deployment of cloud computations across different services, and a resource abstraction layer that provides a unified interface to these services, therefore hiding their low-level differences and simplifying the planning and deployment of the computation. We implemented Conductor and integrated it with the Hadoop framework. Our evaluation using Amazon Web Services shows that Conductor can find very subtle opportunities for cost savings while meeting deadline requirements, and that Conductor incurs a modest overhead due to planning computations and the resource abstraction layer.

1 Introduction

Cloud computing gives programmers access to instantaneous, and practically unlimited computational resources. This allows users and organizations to adapt the computational power they use according to their needs, without requiring them to invest in IT infrastructure. This ease of scalability has made cloud computing popular among end users and a subject of excitement in research and industry. Users have the opportunity to transfer computations into the cloud, enabling applications that were previously impossible or too expensive to perform locally.

These new opportunities, however, bring new challenges. In the past, organizations invested in building and maintaining an IT infrastructure. Given that investment, they could estimate how long a certain computation would take (or its feasibility). In the new cloud computing era, however, it is possible to spend an almost unbounded amount of money on computational resources. This changes the nature of the equation, since organizations can balance the monetary cost of a computation with how long it takes to complete it. Ideally, a customer could invest the exact amount that is needed to complete the required computation within the preferred deadline.

The situation is complicated by the fact that cloud computing providers offer many different services. For example, EC2 currently provides eleven different types of virtual machine instances, and it is unclear how a computation's performance will change if run on different instance types. In addition to the rental of a virtual machine, cloud providers also offer a variety of storage options, in addition to the storage available from the rented virtual machines. Finally, cloud providers charge for data transfer across different systems, particularly between the cloud and the outside world. These factors make it hard to calculate the exact cost of a cloud deployment. Furthermore, the need to account for the possibility of failures (and the cost for recovering from them) and the emergence of spot markets, which allow bidding for resources, aggravate the complexity of making best use of cloud services.

This paper presents Conductor, a system that enables cloud customers to make better decisions about which cloud services to select, and orchestrates the execution of MapReduce computations on the cloud automatically. Conductor therefore frees the customer from having to understand the trade-offs between different services, devising an optimal execution plan, and deploying that plan. For a given MapReduce computation, Conductor lets customers specify optimization goals, such as minimizing monetary cost or completion time, and leverages automated optimization techniques to determine an execution plan that best meets these goals. The system then deploys the plan by invoking the appropriate cloud services at various points in the execution and migrating data among them. Finally, at deployment time, Conductor detects deviations from the expected plan, such as those due to mispredictions of job performance or spot prices, and adapts by recomputing the plan and adjusting the deployment.

[1]Currently at Google Inc.
[2]Currently at CITI/Universidade Nova de Lisboa

The design of Conductor addresses several interesting technical challenges with novel techniques. For example, Conductor was able to cope with the heterogeneity of cloud services, which sometimes combine both a storage and a computation service under the same interface, by designing a resource abstraction layer that separates storage from computation and provides a unified interface to every service of the same class. This abstraction is important to make the planning stage feasible by only needing to consider the generic abstractions offered by the resource abstraction layer. Furthermore, the abstraction layer enables the deployment of computations without the need to worry about lower-level interface details of each specific service.

We implemented a Conductor prototype, which comprises several components: a module for determining the optimal plan for deploying MapReduce jobs, a resource abstraction layer that maps the unified computation and storage interfaces to the suite of services offered by Amazon Web Services, and an extension to the Hadoop framework that interacts with both the planning module and the resource abstraction layer. Our evaluation shows that Conductor's automatic management succeeds in finding and deploying efficient execution plans, and discovers non-trivial opportunities for cost and time savings, while incurring a modest overhead.

The remainder of the paper is organized as follows. In Section 2 we lay out today's challenges when using cloud services and state the goals of our system. In Section 3 we overview how Conductor automates the deployment of computations and in Section 4 we describe how we can formally model MapReduce computations to automatically determine optimal deployment plans. In Section 5 we describe the design and implementation of Conductor, and in Section 6 we present evaluation results with our prototype. We present an overview of related work in Section 7, and conclude the paper in Section 8.

2 Problem Statement

In this section we detail some of the challenges that cloud customers face and describe the goals of Conductor.

2.1 Challenges

The following are several examples of challenges that arise when deploying a computation in the cloud.

Service and provider diversity. Cloud customers must choose among a variety of services with different price and performance characteristics. For example, for its EC2 service alone, Amazon offers eleven different types of VM instances. Furthermore, the diversity of these offerings is increasing, as new providers emerge and existing providers introduce new services.

Hybrid deployments. A special case of provider diversity is when cloud customers make use of their own local infrastructure, which can be augmented by the use of cloud services. Local infrastructures have different characteristics from cloud services, namely that the use of a local infrastructure does not incur additional costs, but provides access to a limited amount of storage and computational power.

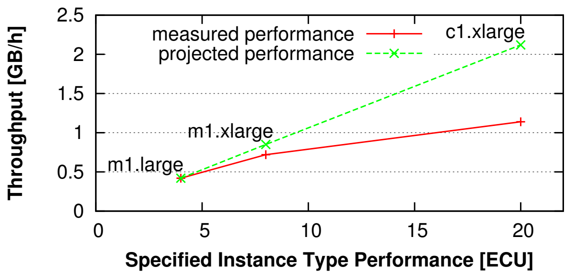


Figure 1: Specified and measured performance for three different EC2 instance types.

Dynamic pricing. The pricing for cloud services can vary over time as providers adjust their pricing models, or when new providers join the market. In addition, spot markets for cloud computing services have been recently introduced in EC2, bringing new opportunities and challenges. In particular, customers may try to use predictions of the evolution of spot prices to obtain cost savings, but may also have to adjust their choices at deployment time in case their predictions are not met.

Mispredictions. To estimate the cost of alternative deployment strategies, customers need to predict the performance characteristics of different services. This is challenging for several reasons. First, the information that is available about these characteristics can deviate from the performance that is actually observed. To illustrate this, we measured the performance of various types of EC2 instances, and compared it to the estimated performance that Amazon reported in terms of a unit they call ECU.[1] The results in Figure 1 show a consistently increasing throughput divergence between the projected and measured application performance. Second, the performance characteristics of a given cloud service can vary dramatically over time [20]. For instance, network throughput might drop due to congestion, not only within the data center, but also on the path between the user and the cloud provider. Also, since often multiple virtual machine instances are hosted on a single physical machine, a level of interference among virtual machines that is higher or lower than normal can lead to degraded or improved performance, respectively.

Faults. Cloud providers have also started to offer services with different reliability characteristics, for instance, with discounted prices for storage services with

[1]For this simple experiment, we use the same setup and application that is used in our evaluation in Section 6, and we configured Hadoop to fully exploit the parallel processing capabilities that each instance type offers.

lower replication factors. These reliability levels are of particular importance in long-running computations, or computations that store intermediate results in these storage services. An example of this are Pig [13] programs, which compile down to multi-staged MapReduce computations, in which the result of one stage is used as the input to the subsequent stage. In this case, when intermediate results become unavailable due to data loss, they must be recomputed by re-executing all previous stages. Therefore, the cost of this recovery depends on the number and complexity of the previous stages, and generally increases as the computation progresses, making more reliable storage options more and more useful [9].

Tightly coupled data and computation. A computation that is deployed on the cloud will make use of several types of resources, namely CPU, storage, and bandwidth. Even though cloud computing providers offer separate services and pricing options for some of these, like storage, most services end up tightly coupling these various categories. For instance, compute services like EC2 associate a virtual disk with each VM instance, which can be used for storage. Also, when performing a computation, one must take into account the cost of transferring the input data to where the computation is performed. This tight coupling complicates the task of deciding which services to use in several ways: it may hide opportunities for making use of resources, such as taking advantage of virtual disks to avoid having to pay for S3 storage, and it precludes simplistic resource management approaches, such as always using the cheapest offering – such a strategy could lead to increasing the overall cost or the completion time, e.g., because of the cost of transferring the data between computation and storage locations.

2.2 Goals

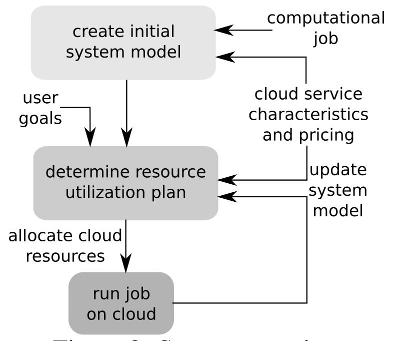
Our goal is to build a system that overcomes these challenges by automating the process of choosing which cloud services to make use of, and by deploying computations on the cloud according to that choice.

Aside from addressing the aforementioned challenges, the system should ideally meet the following goals.

Transparency. Customers should obtain the benefits of the system without having to modify their computations. In particular, they should be able to leverage different types of services without having to adapt their applications to the interface that is provided by that service.

Efficiency. The customer should be able specify certain goals like minimizing cost or execution time, and the system should not only find a good solution according to those metrics, but also impose low overheads both in the planning stage and also at deployment time.

Adaptivity. The system must be able to react at deployment time to mispredictions or changes in the characteristics of the deployment, so that user goals are not jeopardized by these events.


create initial system model
computational job
cloud service characteristics and pricing
user goals
determine resource utilization plan
update system model
allocate cloud resources
run job on cloud
Figure 2: System overview

Flexibility. As cloud services keep evolving, it should be easy to incorporate new services or modifications to an existing service into the system, to allow customers to rapidly take advantage of them.

3 System Overview

To address the aforementioned challenges and goals, we built *Conductor*, a system that simplifies planning and deployment of jobs on the cloud by choosing which services to make use of, according to customer-defined goals, and deploying computations on the cloud according to that choice. In this section we present an overview of the design of Conductor.

As a starting point, a customer outsourcing a computation provides Conductor with the following input: (1) a computation to be executed in the cloud, (2) a set of cloud services that could be used for executing the computation, and (3) a set of goals to optimize the execution for (e.g., minimizing execution time for a given budget).

Given these inputs, Conductor starts by finding an execution plan that best meets the goals specified by the customer. Once the plan is devised, Conductor deploys and executes this plan, and, if necessary, adjusts it to changing conditions, such as variance in network performance due to congestion or degraded virtual machine instance performance due to interference with other instances on the same physical machine.

At a high level, this functionality is achieved through the following sequence of steps (as depicted in Figure 2).

1. Model both the computation and the set of services available from cloud computing providers, their cost, and their performance.
2. Automatically determine an optimal execution plan by using a solver.
3. Deploy the planned execution and monitor the execution to identify conditions that constitute possible deviations from the original model.

4. Upon detecting deviations, feed the new conditions back into the model, compute a new plan, and alter the deployment accordingly.

The next sections detail each of these steps.

4 Modeling Computations and Services

We model computations and the service offered by multiple cloud providers using dynamic linear programming. We chose to use dynamic linear programming because the pricing schemes of cloud services as well as the performance of many data-parallel distributed computations can be expressed as linear dependencies. Furthermore, powerful tools to solve linear programs are available. However, not every computation can easily be modeled using dynamic linear programming, and hence, we had to restrict the domain of our approach. Next, we explain this restriction, and, in subsequent sections, we detail how we model services, computations, and costs.

4.1 Restricting Computation Types

In order to automatically create a linear programming based model for a given job, it is necessary to know what are the individual processing phases, their computational cost, and the data flow patterns among them. Since this is difficult to predict without analyzing the computation, we restrict ourselves to a specific class of computations, namely MapReduce jobs [2]. MapReduce is very generic and has gained widespread adoption, and therefore, by focusing on this model, we cover a very broad and increasingly important set of large-scale computations. Furthermore, MapReduce computations follow a predefined data flow pattern, which makes it feasible to build a generic dynamic linear programming model. Our description assumes the reader is familiar with MapReduce; the original MapReduce paper supplies the necessary background [2].

An alternative to restricting to MapReduce would be to let programmers manually specify these job characteristics, as proposed by other work [5]. Another possibility would be to focus on recurring jobs, where the first run would be monitored to extract the model that would be used in subsequent runs. The core of our system would not have to be changed to accommodate these methods.

4.2 Modeling Cloud Service Offerings

One of the challenges in choosing the best set of services to make use of stems from the fact that each service may coalesce resources of different types, namely storage and computation, which we then need to consider separately when determining the best set of services to make use of. To address this, our model for cloud service offerings breaks down each service into the following three separate types of resources, a subset of which may be provided by that service: computation, storage, and communication. This allows us to have fine-grained control over which services to utilize according to the application needs for each of these resources.

Formally, the model for cloud service offerings considers a set of m distinct cloud services (e.g., Amazon EC2, Amazon S3), $F_1, ..., F_m$, that provide a set of resources of different types. The idea behind our model is to explicitly consider the storage and computation capability of each service, and model communication implicitly. This is because, in contrast to storage and computation, communication cannot stand on its own and be allocated independently, but instead always *connects* other resource instances, which can either be storage or computation. Each service is also associated with a certain price that the customer is charged for using it. For a service that has different prices for allocating a resource for the first time and maintaining a resource, we additionally annotate the communication resources connected to that service to reflect this pricing scheme. For instance, a cloud storage service might charge customers for storage capacity consumed in a period of time, and additionally for network traffic and I/O operations when uploading data to (or downloading from) the service. In this case, in our model the storage service is annotated with the time-based storage cost, and the per-use cost is modeled as a communication cost. Modeling the per-use cost is possible in our case since we can precisely control how data upload and download is mapped to individual I/O operations and how much data is transferred in each operation on average. Thus, given the average amount of data transferred per I/O operation, we can translate the per-operation costs to per-byte costs and incorporate them as communication costs in our model.

Conductor generates the model automatically from a description of cloud service offerings that contains information about service cost, performance characteristics and other properties. In Figure 3 we show a simplified example of a service description in a simple, human-readable XML-based format that Conductor takes as input. These descriptions of public cloud services could be published by the providers themselves or by third parties, while a user would only have to manually specify his privately owned resources (if any).

4.3 Modeling MapReduce Computations

Next, we walk through the successive steps of MapReduce computations to explain how we model them [17]. Unless mentioned otherwise, all variables in our model are positive. We express the execution as a sequence of discrete time intervals such that for each interval t the model contains the actions (e.g., process or transfer data) that can be performed in that interval. An important prac-

```
<resource>
    <property="name">
        <string> S3 </string>
    </property>
    <property="cost_get">
        <double> 1.0E-6 </double>
    </property>
    <property="cost_put">
        <double> 1.0E-5 </double>
    </property>
    <property="cost_t_store">
        <double> 2.08333332E-4 </double>
    </property>
    <property="can_compute">
        <boolean> false </boolean>
    </property>
    <property="storage_capacity">
        <int> -1 </int>
    </void>
</resource>
```

Figure 3: Simplified example of the XML-based description of the S3 storage service.

tical aspect of this model is that, to limit the size of the model that is generated, we always set an upper bound T on the time to finish the computation. T is expressed in terms of number of time intervals, which are the granularity of the execution progress. For instance, one interval could correspond to one hour of runtime.

Input to Map phase. To execute the *Map* phase, the input data from the source storage has to be uploaded to a storage service in the cloud for processing. The upload is modeled in a time-step fashion for all T intervals. For each interval t, the source storage contains $source_t$ amount of data that wasn't uploaded yet and $upload_{(i,t)}$ denotes the amount of data uploaded from the source storage to the storage service F_i. All the data uploaded by time t (denoted by $storeIn_{(i,t)}$) will be stored in F_i until the execution phase is finished. Data storage and upload is *flow preserving*, which we express by the following constraints:

$$\forall i,t:\ source_t - \sum_{i=1}^{m} upload_{(i,t)} = source_{t+1} \qquad (1)$$

$$\forall i,t:\ storeIn_{(i,t-1)} + upload_{(i,t)} = storeIn_{(i,t)} \qquad (2)$$

The available upload speed can be expressed in the model by adding a constraint that restricts the total amount of data that can be uploaded.

Data processing. Next, the uploaded data is processed, and the result is stored. Similar to data upload and storage, we model the actual processing per time interval t: In interval t, the uploaded data $storeIn_{(i_1,t)}$ in storage service F_{i_1} can be processed by a computational service and then the result $storeOut_{(i_2,t)}$ is stored at F_{i_2}.

The amount of data that is processed in each time interval t is bounded by the number of computing nodes that we choose to run during that interval. Also, we can only process input data in the cloud that has already been uploaded. Let $proc_{(i,t)}$ denote the amount of data which is processed by cloud service F_i in time interval t. We can therefore represent the constraints for computations as follows:

$$\forall i,t:\ \sum proc_{(i,t)} \leq nodes_{(i,t)} \cdot capacity_i \qquad (3)$$

$$\forall t:\ \sum_{t'=1}^{t} \sum_{i=1}^{m} proc_{(i,t')} \leq \sum_{i=1}^{m} storeIn_{(i,t)} \qquad (4)$$

Here, $nodes_{(i,t)}$ denotes the number of computing nodes rented in interval t from computing service F_i, and $capacity_i$ denotes the processing capacity of a single node for F_i.

Reduce phase. The *Reduce* phase is modeled in a similar way to the Map phase, except for the fact that we do not need to consider the data upload stage, since the *Reduce* phase takes the result of the *Map* phase as the input. Hence, in our model, in each time-step t we add possible transitions from the output storage of the *Map* phase $storeOut_{(i,t)}$ to the input storage of the *Reduce* phase $storeIn_{(i,t+1)}$.

With the current formulation, the Reduce phase can start without the Map phase being complete, which is not allowed by the MapReduce model. We enforce that the two phases do not overlap by specifying that the amount of data flowing to the next phase has to be either 0 or the full output data. We specify this property as a linear programming constraint using a *semi-continuous* variable, that can hold either 0 or the full output data size. After combining the two phases, we model the download of the final result from the output storage of the *Reduce* phase by adding transitions to the destination storage.

4.4 Execution Cost

The model must also capture the monetary cost of running the computation. The monetary cost of each phase can be expressed as the cumulative sum of the cost incurred in each interval over time T. For time interval t, the cost can be expressed as the sum of the cost incurred for uploading the data, processing the data and storing the result in a storage service. We calculate the cost for each time interval based on the amount of resources consumed per cloud service. For instance, the computation cost in time interval t is the number of machine-hours used in this interval multiplied by the price per machine-hour. Formally, we express the total monetary cost over T as follows:

Let $y_{(i,t)}$ be the number of units of cloud service F_i purchased for time interval t, and let b_i be the price per unit for F_i. The total cost C for such a configuration is

$$C = \sum_{t=1}^{T} \sum_{i=1}^{m} (b_i \cdot y_{(i,t)}) \qquad (5)$$

Note that this monetary cost, as well as other characteristics captured in our model such as execution time, can be used in the objective function for optimization. Since no negative amount of resources can be purchased, we automatically have the constraint $\forall i, t: \; y_{(i,t)} \geq 0$.

4.5 Data Migration

Since we consider multiple storage services in our model, we may choose to migrate data between them during the execution. We include migration by adding transitions in each time interval t from $storeIn_{(i_1,t)}$ to $storeIn_{(i_2,t+1)}$. These transitions express migrating input data from the storage services F_{i_1} to F_{i_2}. Similarly, we add transitions for migrating the output data $storeOut$ in each time-step. Note that the transitions for data migration go from one time-step t to the next one $t+1$, rather than staying within the same time step. This allows us to express that data migration is not completed instantly. The cost for migration can be added to the storage cost per time-step.

4.6 Resource Overlap

In our previous explanation of the model, we have assumed that each service provides only a single type of resource, either storage or computation. However, in practice, services can provide both types (and potentially other types) simultaneously. For instance, we can opportunistically store data on the virtual disk of running VMs, leveraging this spare resource at a low extra cost.

Our model accommodates this overlap of resources easily, since it distinguishes cloud services from the resources they provide. Thus, in addition to the pricing and performance characteristics we already specify for each cloud service $F_1, \ldots, F_m$, we also specify the quantities of other resources $R_1, \ldots, R_n$ each of the services offers. For instance, in this model a pure storage service like Amazon's S3 will provide only storage resources while instances of Amazon's EC2 service provide both computation and storage resources.

4.7 Dynamic Pricing

Recently, Amazon started offering spot market pricing, where customers bid the maximum price they are willing to spend to have access to unused Amazon EC2 capacity, thus paying a price tag that reflects the current supply and demand. Furthermore, Amazon allows customers to have access to the history of spot prices, so that customers can try to predict how spot prices will change and develop a bidding strategy.

We thus extend our model to include dynamic pricing in spot markets. Given our model where computations are divided into discrete time-steps, spot prices can easily be incorporated by setting the price of this service in each time-step to an estimated spot price. These estimates could potentially be derived by extrapolating past pricing patterns. In our evaluation in Section 6.5 we leverage a simple method that uses the maximum spot price of the last n hours as a basis to compute a bid. More elaborate methods [1] or methods for analyzing stock market trends could also be leveraged. However, predicting spot prices is a challenging problem in its own right and beyond the scope of this work. For the sake of simplicity, we assume *some* predictor that can produce estimates for future pricing. Let $E[b_{(i,t)}]$ denote the estimated price per unit of cloud service F_i for time interval t. Thus, the modified total cost C' can be expressed as follows:

$$C' = \sum_{t=1}^{T} \sum_{i=1}^{m} (E[b_{(i,t)}] \cdot y_{(i,t)}) \tag{6}$$

4.8 Solving

For processing the linear program and computing an optimal execution plan, we dispatch the generated linear program to the CPLEX solver. Although the solving time is usually on the order of seconds (see Section 6.6), it is possible that in certain cases CPLEX takes significantly longer to compute the optimal solution. In such cases, a potentially non-optimal, but feasible solution can be found much faster. To avoid long delays on the deployment of jobs submitted by users, instead of waiting for the optimal solution, we bound the solving time to three minutes and use the best solution that was computed so far.

5 Job Deployment

Once Conductor finds an optimal execution plan for a model of a job and available resources, it deploys the plan by instantiating the appropriate cloud services. We next present the design of this component of Conductor.

5.1 Programming Abstractions

The deployment of a computation plan is complicated by the fact that different services may have different storage and computation interfaces, incompatible semantics, or that sometimes storage and computation are bundled together. Conductor overcomes the differences between the services by providing a uniform interface to applications. In particular, Conductor provides abstraction layers for the two basic resource types: storage and computing resources. For services with bundled resources like EC2 instances, the abstraction layers for these two different resource types allow using storage and computation independently. These abstraction layers also enable Conductor to transparently manage the resources according to the execution plan. Furthermore, the abstraction layers hide the complexity of supporting and managing the resources from the application, as depicted in Figure 4.

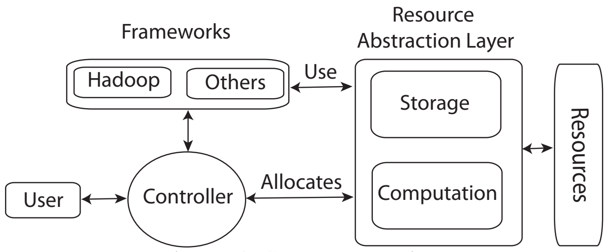


Figure 4: System overview.

Storage. The primary goal of Conductor's storage system is to provide an abstraction layer that enables applications to transparently utilize multiple different storage services as backends, and manage this usage automatically. For instance, by using Conductor's storage system, applications can transparently use both S3 and the local hard disk of virtual machines to store different parts of the data. This storage can be accessed by a client that hides from the user how and where data is stored on the backends.

We implemented Conductor's storage system as a distributed key-value storage service. The key-value interface is generic enough to support other abstractions built on top of it. For instance, there are already multiple file system implementations built on key-value stores [12]. Also, many applications and frameworks, including Hadoop, support Amazon's S3 storage service. Since S3 and Conductor's storage system provide similar interfaces, Hadoop is able to use Conductor's storage system seamlessly.

The central component in Conductor's storage system is the *namenode*, which provides a directory service for data, and manages upload, replication and migration of the data as per the execution plan generated by the controller (described in Section 5.2). The namenode maintains a mapping from file block identifiers to their locations in Conductor's storage system. These location records contain information specific to the storage backend on which a file block is stored. For instance, for a file block stored on a node's local hard disk, the location record would indicate the type of storage and include the addresses of the nodes storing that file block. (We replicate blocks in more than one node for fault tolerance and performance.)

The implementation of each storage backend is specific to the storage services it utilizes, and maps the semantics of each service to the target key-value store semantics. For instance, the local disk storage backend uses a storage daemon running on each participating node. This daemon uses Berkeley DB to store key-value pairs locally on disk. The data stored on each node can be accessed through a protocol with put, get and delete queries. For the S3 backend, in contrast, the client uses the generic S3 client APIs.

To access data in Conductor's storage service through the uniform interface provided by the storage client, the client first queries the namenode for a set of locations for the data block. In case the namenode returns multiple locations, the client fetches the file block from the closest location (by ping time) using the logic that is specific to the storage backend given in the location record.

As an optimization, when computation and storage components are co-located on the same node, Conductor allows the computation to make use of local storage without going through the namenode. For reading data, this is done by directing requests to the local storage daemon directly, which can either succeed and proceed in a very fast manner, or fail and fall back to the normal read operation, in which case we additionally install a cached copy of the data on the local node. For writing data, the optimized write is always performed locally, and then the namenode is notified so that it can transfer data to the appropriate nodes in the background.

Computation. For computation it is more difficult to create an abstraction layer for different services. For instance, it is easy to provide a MapReduce computation service (like Amazon's Elastic MapReduce service) on top of a virtual machine abstraction, but the converse is not true. Since we restrict ourselves to MapReduce computations, our abstraction of a computation resource is an instance that is capable of participating in a MapReduce computation. In particular, we only require that computation resources can be configured and automatically set up to join a Hadoop cluster and participate in MapReduce job execution. This allows us to implement this abstraction on top of different types of services, from low level VM rental to local cluster resources. For instance, in the case of Amazon's EC2, this is achieved by building a pre-configured machine image which is used by Conductor to automatically allocate EC2 instances according to the deployment plan and have them join a Hadoop cluster.

5.2 Job Controller

The job controller is a central component of the design of Conductor. It orchestrates the job execution by generating a plan to best meet the user's goals and deploying it using cloud services.

After submitting a MapReduce job to Hadoop, a user starts Conductor's job controller to manage the execution of the job. The job controller automatically generates a linear programming based execution model with the given job characteristics and resources available as described in Section 4. The model, together with the user's optimization goals, is then processed by a solver to generate a deployment plan. After the controller receives the resulting plan, it deploys it accordingly and monitors the execution progress. Deployment decisions concerning how to handle and store data (e.g., where and when to upload and store what data) are forwarded to the storage service (described in Section 5.1), which then triggers upload and replication accordingly. Deployment decisions about the actual processing are handled by the job controller by allocating the planned number of nodes through a service-specific interface (e.g., in case of EC2, Amazon's AWS client library) and setting them up to join the computing cluster.

In order to allow a seamless interaction between Hadoop and Conductor, we extended Hadoop in several ways, as we explain next.

5.3 Hadoop Extensions

We adapted Hadoop version 0.20.2 to support Conductor's automatic management functionalities.

Location-aware scheduler. The original Hadoop scheduler makes decisions that may conflict with the execution plans determined by Conductor, thus resulting in higher cost or performance degradation. In particular, the Hadoop scheduler tries to schedule tasks on the nodes that also hold the respective input data block, and, in cases where locality cannot be exploited, it schedules tasks on non-local nodes and reads their input over the network. This flexible scheduling of tasks conflicts with Conductor, as it may violate the deployment of the execution plan generated by the controller. For example, fetching the input data from a remote site when not specified in the plan could congest network links, hinder other data transfers, or result in transfer costs that were not considered during the optimization phase. Therefore, to accurately deploy an execution plan, we must override the flexible scheduling policies of the Hadoop scheduler. In particular, we ensure by data migration and replication that data is always locally available to the task or stored on a remote location specified in the plan when it is assigned for execution by the scheduler.

The modified scheduler is integrated into Hadoop and we normally run it on a node under the customer's control. To deploy the plan accurately, the scheduler maintains task queues for each computing resource (e.g., EC2) containing the tasks that are runnable for that resource. The scheduler sets tasks runnable when their input data is either stored locally to that resource or on a different storage resource specified in the plan. For instance, depending on the plan, a task is set runnable on EC2 nodes when its input data finishes uploading to EC2 nodes or to the S3 storage service. The scheduler then assigns runnable tasks from the corresponding queues to nodes. This mechanism ensures that during scheduling no actions are performed that were not considered in the plan, which might have negative impact on runtime or cost.

Storage system. The second extension to Hadoop is to add support for Conductor's storage system. This support is required for Hadoop jobs to process input data stored on Conductor's storage system and write output data to it. Hadoop supports multiple storage options via *file system drivers* that implement a file system abstraction. In order to make Conductor's storage system usable by Hadoop, we implemented a file system driver that translates file system specific calls (e.g., open, close, read, write) into the key-value store operations (e.g., get, put, delete) that are supported by Conductor's storage system.

In our implementation, we split files into smaller *chunks* that are stored as key-value pairs in Conductor's storage system. Additionally, for each file we store *inodes* that list the chunks that constitute the file content. Our implementation reuses to a large extent the Amazon S3 file system driver, since S3 has a similar key-value storage interface. In contrast to the S3 driver, which does not allow any locality in scheduling tasks, the driver for Conductor's storage system implements the functionality required by the Hadoop scheduler to perform location-aware scheduling. More precisely, we provide methods for the scheduler to retrieve the location of a task's input data, and, based on that information, set it to runnable. The driver also interacts with Conductor's storage system to provide hints about which data block should be uploaded or replicated with higher priority.

5.4 Adapting to Dynamics

The job controller monitors the execution progress. If the observed performance for a particular resource (e.g., EC2 instances) significantly deviates from the expected characteristics upon which the model and the deployment plan was based, the job controller adapts the deployment by creating an updated model, recomputing the plan, and deploying it accordingly.

In a similar way, Conductor reacts to other system dynamics that might change during runtime, such as dynamic pricing for resources in spot markets. Conductor re-creates a model based on the current system state and the properties of the resources, including the changed ones. Similarly to the initial model, this model is transferred to the solver daemon to determine an execution plan and deploy it.

6 Evaluation

In this section we evaluate our Conductor prototype by using it to deploy several computations on the cloud using Amazon's Web Services (AWS) in scenarios that can be difficult to handle manually or require non-obvious deployment strategies.

Our evaluation tries to answer the following main questions: (1) Can Conductor realize potential cost and time savings when deploying MapReduce jobs, both in cloud-only and hybrid deployments? (2) Can Conductor adapt to unexpected conditions at deployment time, including the unpredictability of spot market prices? (3) What are the overheads introduced by Conductor?

6.1 Experimental Setup

In all experiments, the plan generated by Conductor exclusively makes use of large instances (`m1.large`) for processing on Amazon's Elastic Compute Cloud (EC2). These instances are equipped with 7.5GB of memory, a 850GB virtual hard disk, and 4 EC2 Compute Units, where one Compute Unit is equivalent to a 1.0-1.2GHz AMD Opteron or 2007 Intel Xeon CPU. In addition to the large EC2 instances, we also allow Conductor to use extra large EC2 instances (`m1.xlarge`). However, in the scenarios we consider, the extra large instances are never chosen in the generated deployment plans since they offer a cost-performance ratio that is slightly worse than for large instances. For hybrid deployments, we additionally use a local cluster of five machines, each equipped with an AMD Athlon64 dual core CPU running at 2GHz and 2GB of memory. Additionally, some experiments make use of S3 for storage.

The application we use for our evaluation is a k-means clustering analysis. We use the k-means clustering implementation in MapReduce that is available as part of the Apache Mahout package. The input to the job consists of 40 million randomly generated points, summing up to 32GB of data. Additionally, we use a set of 10 thousand reference points for the clustering process. For this application, the large EC2 instances we used and our local cluster nodes both achieved an average processing throughput of 0.44GB/h per node. Our approach can be applied to other applications and resources as well when their characteristics are specified.

Unless mentioned otherwise, the network bandwidth between the customer and the cloud is set to 16MBit/s (2MB/s) and the client has a predetermined deadline for job completion of 6 hours. In all experiments, the input data and the Hadoop Jobtracker were located on a node in our local cluster, and the output was also downloaded to our local cluster. We used the prices of Amazon's AWS as of July 2011. For tracking the cost of cloud resource usage in each experiment, we instrumented our prototype implementation to account for all operations over cloud resources. We chose this internal accounting approach over Amazon's accounting because it enabled us to track the per experiment cost and at a much finer granularity.

6.2 Savings in Public Clouds

First we evaluate Conductor's ability to deploy an execution plan that realizes potential cost and time savings in a scenario where the customer deploys a computation entirely on the cloud.

In this scenario, the customer has several options for deploying the computation using AWS, which we test in our experiments:

- *Hadoop S3.* Upload data to S3 and then instruct a Hadoop cluster running on EC2 instances to access data directly from S3.
- *Hadoop upload first.* Upload data directly to single EC2 instance running HDFS. Upon completion, start more EC2 instances to join the cluster and use HDFS for inputs and outputs.
- *Hadoop direct.* Set up the HDFS cluster on the client side, and instruct the EC2 instances to read and write to this HDFS cluster.

Interestingly, all of these options are described in the Hadoop or AWS documentation, which further strengthens our motivation that there often does not exist a clear choice of how to deploy computations in the cloud.

Figure 5 shows the monetary costs for different deployment options and Figure 6 shows their overall job completion time. The first four bars compare Conductor's cost and performance to the three deployment options listed before. In this case, Conductor determined it should only use EC2 instances for storage, and the adequate number of EC2 instances to be 16. Therefore, in the *Hadoop direct* run, we also use 16 EC2 instances. For the two configurations with a distinct upload phase before processing, we use 100 EC2 instances. In the runtime comparison in Figure 6, *streamed processing* denotes the combined time required for processing the data and retrieving it from the respective storage as it is consumed, without a distinct upload phase.

From these results we make two main observations. First, that the total cost and completion time can vary significantly between the different deployment options. For certain deployment options (e.g., *Hadoop S3*) the service pricing models can result in an unexpectedly high total cost. In the case of *Hadoop S3*, the actual processing was finished in little more than one hour, but two full hours are charged for each allocated instance, resulting in a total cost roughly two times higher than for the other options. The second observation we make is that Conductor succeeded both in obtaining a cost that is very close to the cheapest alternative, and in meeting the required completion deadline. Note that the fact that Conductor only performs slightly worse than the fastest alternative is a positive outcome, given that we are comparing the performance of our implementation mostly driven by a single graduate student with Hadoop's highly optimized production code. We analyze the main overheads introduced by Conductor later in the evaluation.

Another advantage of Conductor is that it not only helps determine the best deployment scheme but also helps choose the right deployment parameters, which may be even more difficult to set than just determining, e.g., whether to use S3 or not. To show this, we validate

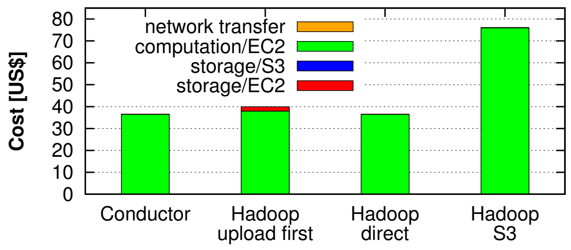


Figure 5: Monetary cost for running the job for various deployment options solely in the cloud.

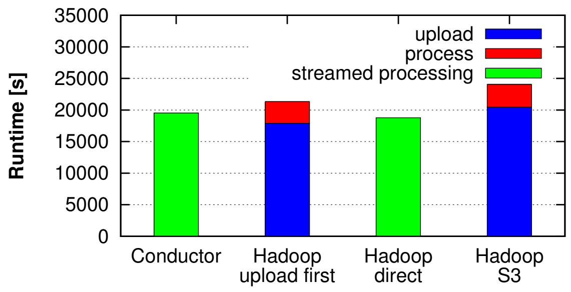


Figure 6: Job completion time of various deployment options solely in the cloud.

whether Conductor made a good decision regarding the number of EC2 instances to reserve. Therefore, we reran the experiment with five more and five less EC2 instances than those chosen by Conductor (11 and 21 instances, respectively). The results are shown in Figure 7. These show that slightly increasing and slightly decreasing the number of EC2 instances that are allocated leads to either a cost increase or missing the deadline, respectively. This validates our points that understanding the characteristics of all possible deployment options and making the right choice for a particular application scenario can be challenging, and that Conductor performs this choice automatically while incurring in modest overhead.

While this first experiment already shows some challenges in deciding which cloud services to use, Conductor still ends up resorting to one of the three deployment possibilities we had considered initially. In the next experiment we intentionally designed an even more challenging scenario where multiple different services have to be used at different times to minimize the monetary cost for the execution. To highlight this, we slightly modify the job parameters to use an upload bandwidth of 8MBit/s and a smaller set of reference points such that large EC2 instances process the input at a rate of 6.2GB/h per node.

The experimental results in Figure 8 show that neither storage option yields optimal results when used alone. Instead, the minimal cost is achieved when a mix of S3 and EC2 storage is used for storing different parts of the data at different points in the execution. In this particular example, Conductor first uploads roughly half of the in-

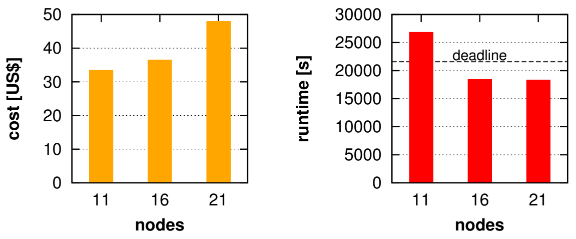


Figure 7: Monetary cost and runtime impact of deviating from the optimal deployment scheme in a cloud-only scenario.

put data to the S3 storage service, and later the remaining data is uploaded to EC2. Once an EC2 node is allocated, it starts processing the input data that was previously uploaded to S3 and the data is uploaded in parallel to an EC2 node. This utilization plan, in contrast to using only S3 or EC2 for storage, makes the best use of the EC2 nodes and the S3 service to achieve a lower monetary cost. This non-obvious resource utilization plan, which would be difficult to determine manually, is found and deployed automatically by Conductor, thanks to both the modelling and optimization phase and the use of resource abstraction layers that allow for seamlessly using the two types of storage in combination.

While Figure 8 only shows modest cost savings when compared to one of the simpler options (storing all data on S3), we point out that the cost savings for a combined solution can be much higher, since (1) the percentage gains we illustrate in the experiment increase with the input data size, and (2) these effects are also sensitive to variations in the pricing structure. Since we did not consider larger data sizes in our experiments due to financial and experiment duration constraints, we determine the potential cost savings analytically, by assuming a different input size and pricing structure. The analytic results in Figure 9 show what happens when we assume S3 storage costs are ten times higher and scale up the input size to 64, 128, and 256 GB. These results show that hitting the sweet spot for utilizing different cloud services has an increasing impact on monetary cost as data size increases, reaching savings of about $1/3$ of the cost for an input of 256 GB.

Note that the optimal fraction of data to store on EC2 when considering 32GB of input data is higher than the optimal fraction when considering larger input data sizes. This effect results from the accounting granularity of EC2 instances: since allocated node-hours are rounded up for billing, Conductor does not immediately shut down allocated instances after the computation is finished, since they are billed until the next full hour anyway. Instead, these instances are used for storage. For larger input data sizes, these rounding effects have much less impact and instances are mostly used for computation and storage simultaneously.

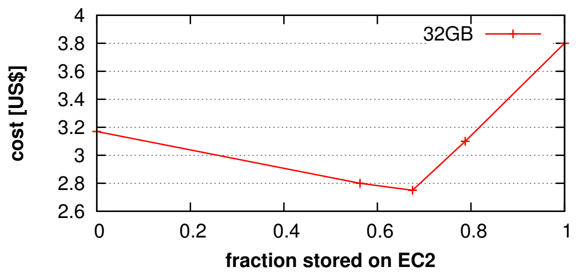


Figure 8: Total job cost depending on where the 32GB of input data is stored. A fraction of 1 (0) stored on EC2 denotes that all input data is stored on the virtual hard disks of EC2 nodes (on the S3 storage service). Conductor determined that in this scenario costs are minimized when roughly two thirds of the data is stored on EC2.

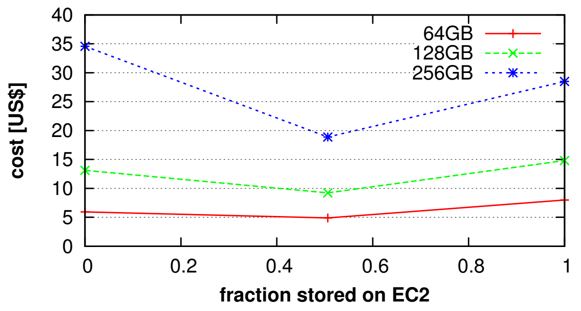


Figure 9: Total job cost depending on where the input data is stored. A fraction of 1 (0) stored on EC2 denotes that all input data is stored on the virtual hard disks of EC2 nodes (on the S3 storage service). Conductor determined that in this scenario costs are minimized when roughly 50% of the data is stored on EC2.

6.3 Savings in Hybrid Clouds

The next experiment determines whether Conductor can realize potential cost and time savings in a scenario where the cloud customer can make use of a local cluster for some of the processing, but this capacity is not enough to meet the prescribed deadline. This local cluster is modeled as just another provider (which is the user himself) that offers a single instance type (which is the machine type in the local cluster). To account for the limited size of the local cluster, we enforce a constraint in our model that limits the number of instances that can be rented.

In this scenario, Conductor determined that data should be stored on EC2 instances, and decided that the right number of EC2 instances to allocate was 16 to meet the deadline of 4 hours. In Figure 10 we show a cost and runtime comparison with an HDFS-based deployment that also allocated 16 instances. The results show that, even if the user managed to guess the right number of instances to allocate, the results that are obtained are very similar to the ones achieved by Conductor. Furthermore, Figure 11 shows what happens if the user under-

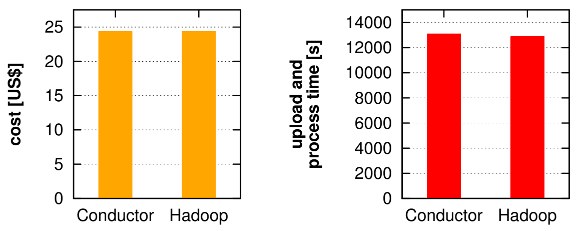


Figure 10: Monetary cost and runtime for running the job with Conductor and Hadoop when leveraging local resources.

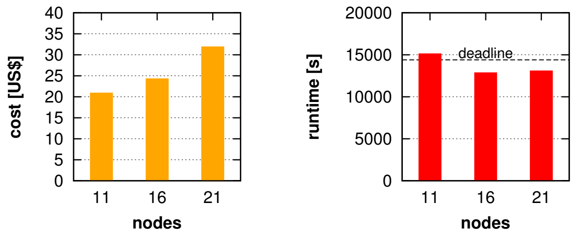


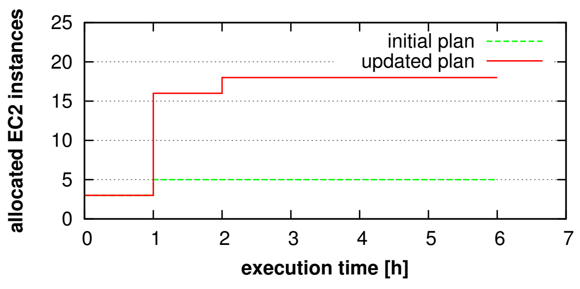
Figure 11: Monetary cost and runtime impact of deviating from the optimal deployment scheme in a hybrid scenario.

estimates or overestimates the number of EC2 instances to allocate. Again, this could lead to either an increased cost, or to missing the deadline.

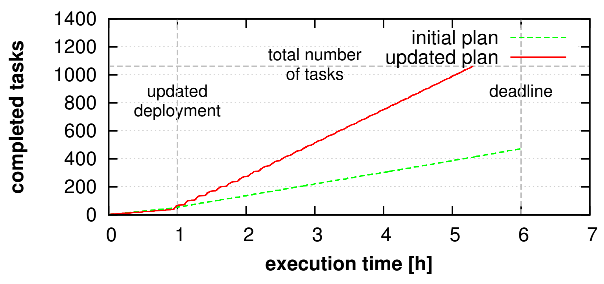
6.4 Adapting to Performance Variations

In this section we present our experimental results to demonstrate how Conductor can adapt to dynamics during application runs.

In this experiment we wrongly assume a processing speed of 1.44GB/h per node when the actual speed is 0.44GB/h. Such a difference between predicted and actual processing rates may result from wrong estimates by the user, but also due to the heterogeneity in cloud node performance [20]. Figure 12 plots the number of allocated EC2 instances and total completed tasks throughout the job execution. In the initial deployment plan, Conductor used 3 EC2 nodes in the first hour of execution and 5 EC2 nodes from the second hour on. This number of nodes would be sufficient to finish the job if the processing speed per node would indeed be 1.44GB/h. After one hour, the job progress monitoring revealed the misprediction, which caused Conductor to update the model and recompute the deployment plan. The new plan is unchanged for the first hour (corresponding to the past execution) but uses 16 EC2 nodes in the second hour and then 18 EC2 nodes from the third hour on. With this updated deployment plan, the job can be finished before the deadline even though the initial plan would have led to missing that deadline. Similarly to performance over-estimation, Conductor can react to underestimation by adapting the deployment and reducing the number of EC2 instances to use.

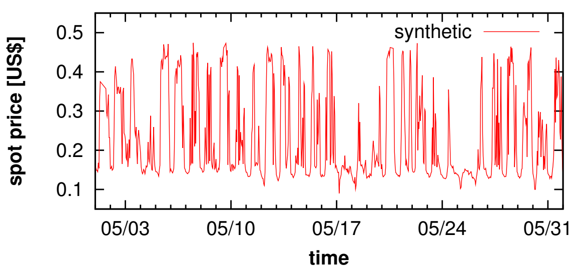


(a) Instance allocation in initial and updated deployment plan.

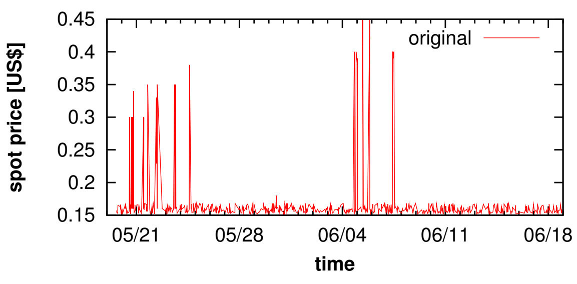


(b) Job progress in initial and updated deployment.

Figure 12: Job progress and Node allocation with initial and updated deployment plan.



(a) Synthetic history generated from electricity spot market.



(b) Original history for AWS EC2 instances.

Figure 13: Spot price histories for AWS EC2 instances of type m1.large

6.5 Adapting to Dynamic Pricing

Next, we evaluate through simulations the monetary savings from integrating Conductor with spot markets.

To drive spot market prices in these experiments we initially intended to use only the EC2 spot price history from AWS. However, we realized that the history of these spot prices did not exhibit any diurnal patterns, as can be seen in Figure 13. As more providers enter the market and spot markets attract more customers, we expect the price to more closely reflect data-center utilization and resource demand, and hence become more predictable. Therefore, we decided to include a second data set in our evaluation for which history can be used as a reasonable predictor. For that purpose, we use the spot price history from an electricity market. This data had to be slightly adapted, namely to make values non-negative (electricity spot prices can be negative), and also to keep the values below the normal price of EC2 instances.

In Figure 14 we present our simulation results for cost savings with Conductor when using spot instances in different settings. In *regular*, only regular instances (without dynamic pricing) are used. In *aws* we use the original spot price history for Amazon's EC2 instances, while in *el* we use the synthetic history generated from electricity prices. *-opt* denotes the cost in an optimal deployment case where Conductor can exactly predict spot prices. In the *-pX* settings Conductor uses a simple predictor that uses the past X days of spot pricing history to derive a price prediction. With *-p0*, the predictor assumes the current spot price will not change.

A first observation from these results is that allocating EC2 instances on the spot market can reduce the total job cost compared to allocating regular instances. The average cost savings in both settings range between 50% to 60% – a significant reduction.

A second observation is that the use of a trivial predictor (*p0*) is highly effective in both spot markets, achieving close to optimal cost savings. As the predictor becomes more sophisticated by incorporating more information from the recent past, there are slight improvements when using the electricity prices, namely in reducing the standard deviation of the final cost. Thus in this case the planner can efficiently leverage historic spot data. However, when considering the less regular AWS trace of spot prices, the use of more information from the recent past causes the total price and the standard deviation to go up. This is because there end up existing more situations where the plan decides to wait for a better spot price and at deployment time ends up waiting in vain.

6.6 Overheads

Storage layer performance. We compare the performance of the storage layer offered by Conductor to other storage options in Hadoop, namely HDFS and Amazon's S3. We measure throughput in our experiments, since this is the most important performance metric for MapReduce workloads. We do so by copying 32GB of data (consisting of 64MB files) to the corresponding storage service. To allow for a fair comparison to S3, we ran the measurements on large EC2 instances, where the network bandwidth to S3 is higher, instead of on our cluster nodes.

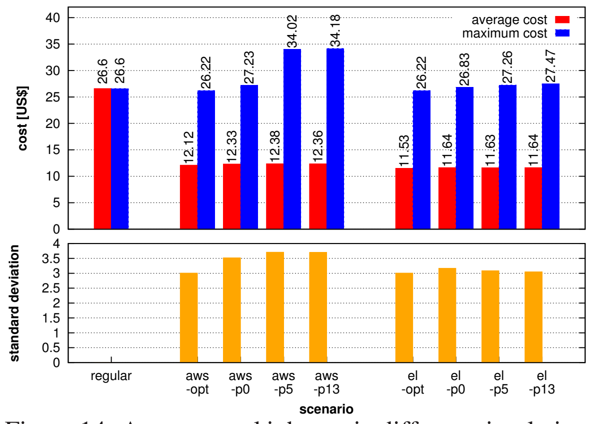


Figure 14: Average total job cost in different simulations.

Conductor's storage service and HDFS were configured with a replication factor of three, and all four nodes were either used as datanodes for HDFS or ran a storage slave for Conductor's storage. In all cases, the data was read from an Elastic Block Store volume. We considered two options for copying the data onto S3: using the integrated S3 support of Hadoop and using `s3cmd`, and a dedicated S3 command line client. The reason for also considering a separate S3 client is that we found a significant performance gap between these two options that cannot be attributed to the S3 storage service, but rather to client implementation specifics: the S3 client integrated into our version of Hadoop used SSL data transfer to S3 by default, which significantly decreased performance.

The results presented in Figure 15 show that the highest throughput was achieved with Hadoop's HDFS. Conductor's storage service exhibits roughly 25% lower throughput than HDFS in our experiments and performs comparably to S3 when using `s3cmd`. The measured throughput of S3 when using Hadoop directly is significantly lower than using `s3cmd`.

The high performance of HDFS was not particularly surprising to us, as HDFS has been actively developed for several years and significant effort has been put into performance optimization. In contrast, in our prototype the main focus was an abstraction layer that could utilize several other storage services. Therefore we believe we introduce an acceptable throughput overhead.

Modelling and Solving. Creating a linear program-based model consisting of all possible actions that could be taken for a MapReduce program and determining an optimal plan are presumably expensive operations. However, in our prototype it turned out that the model creation is very cheap. For all scenarios we considered in our evaluation, the model creation took less than 1 second on a desktop machine with an Intel Core 2 Duo CPU at 3GHz and 4GB RAM. Computing an optimal execution plan from such a model was significantly more expensive. In our experiments, we used the CPLEX 11.2.1 optimizer for that purpose. We ran CPLEX on a node equipped with 8GB of RAM and an AMD Opteron dual core CPU running at 2.6GHz. We configured CPLEX to terminate when a solution is found that is at most 1% off the optimal solution.

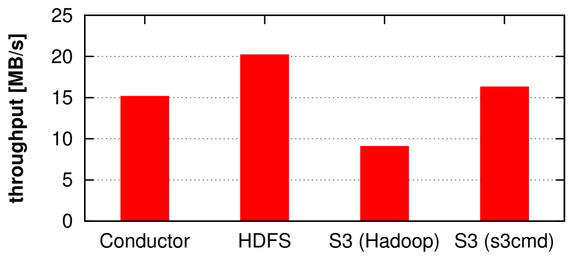


Figure 15: Performance comparison of different storage options.

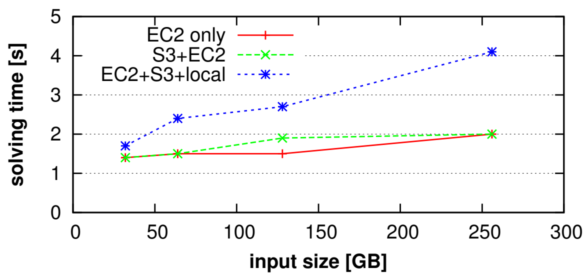


Figure 16: Model solving time for different input sizes and available resources.

Figure 16 depicts the solving time for various input sizes and different resources that are available to run the job. EC2-only means that we assume to have only EC2 available for both computation and storage. In S3+EC2 we also allow the use of the S3 storage service and in EC2+S3+local we can additionally use a local cluster for storage and computation. The results show that the solving times are acceptable, and that adding more features to the model roughly doubles the solving time. Furthermore, we can see that the input data size directly influences solving time. This is because the input size has an impact on the model size (since the input size, together with processing speed and upload speed, gives a lower bound on execution steps to include in the model), which then implies an increase on solving time.

7 Related Work

Our work builds on contributions from several different areas, which we briefly survey.

Resource Management. Automatic resource management has been studied in other contexts. For example, operating systems automatically allocate resources; cluster resource management systems have been proposed [3, 7]; and resource management in Grid computing has also been studied [10, 15]. In cloud computing, resource management presents new challenges that are not addressed by previous work. Namely, as opposed to clusters and grids, a cloud computation has access to infinite resources but must pay an increased marginal cost for each resource it uses. Our system takes these distinguishing features into account, and modeling them is one of the primary challenges of this work. A recently proposed system called Mesos [6] allows different cluster computing frameworks to share a static commodity cluster to improve utilization and reduce data redundancy. Mesos employs a scheduling mechanism in which resources are offered to the different frameworks, and each framework can decide which resources to use and how to use them. Besides targeting a dynamic cloud setting, Conductor differs from Mesos by considering job deployment at task-level granularity, taking into account user-provided optimization goals and the costs of the on-demand cloud resources.

Scheduling. There is a significant amount of research in scheduling jobs in the context of distributed execution frameworks. Quincy [7] is a fair scheduler for scheduling concurrent distributed jobs with fine-grain resource sharing for Dryad. Late scheduler [20] overcomes the performance degradation due to straggler tasks in Hadoop for heterogeneous environments. Delay scheduling [19] strikes a balance between data locality and fairness by employing a lazy approach for scheduling jobs to maximize the locality. The most fundamental comparison point is that Conductor must consider the dynamic allocation of cloud resources as an additional dimension in the scheduling problem, while the aforementioned work can simply assume a static set of resources.

Optimization. Many systems require decisions to be made at runtime from a large set of possible alternatives. Therefore, optimization techniques have been employed to select the best choice dynamically. For example, Rhizoma [18] proposed automating resource allocation for generic applications. Rhizoma uses a specification of resources and maximizes the utility for an application. Although Rhizoma uses cloud computing as motivation, the application they describe is a publish-subscribe system deployed on PlanetLab, where the challenge is to find well-connected, lightly loaded nodes. Unlike Rhizoma, our proposal is geared towards deploying computations on the cloud, without requiring the specification of application resource requirements. We model the problem of cloud resource allocation as a linear program. Modeling other problems in such a way has been done in a variety of fields including systems research [8]. Similarly, a recent proposal [14] seeks to shift computations among multiple data centers based on changing electricity prices in the spot market. Conductor's approach is related, but addresses a different problem of making the best possible use of a diverse set of cloud resources for a specific type of computation, for which a runtime model can be derived automatically. Our own short position paper [16] described high level ideas, which are incorporated in this work, but did not present a complete system.

Hybrid deployment and spot markets. With the advent of public and private cloud infrastructures, there is a demand to utilize both of them. CloudWard Bound [5] makes a case for the hybrid deployment of multi-tier enterprise applications where the infrastructure is partly hosted on-premise, and partly in the cloud. For a given application, CloudWard Bound can suggest deployment plans that leverage cloud resources for some application components, obey user-provided privacy policies and satisfy application latency requirements. In contrast, Conductor focuses on a distributed bulk data-processing framework where it can manage the deployment of processing tasks for individual jobs. Conductor's deployment plans need to consider the varying resource requirements jobs can have throughout their execution, which is less relevant in the context of long-term deployments of enterprise applications that CloudWard Bound targets. Furthermore, data privacy is not the focus of Conductor.

Dynamic allocation of spot instances for MapReduce computations has also been proposed recently [1, 11]. In contrast, we focus on the broader problem of trying to incorporate multiple providers of potentially diverse resources (both from regular and spot markets) to determine a globally optimal resource allocation plan.

Resource exploration. The availability of multiple machine types raises the question of how the different machine characteristics will impact application performance. Accurately predicting application performance when low-level characteristics are known is a challenging problem that has been studied in the past [4]. We consider the problem of resource exploration to be complementary to our work; our approach could directly benefit from resource exploration techniques since we can leverage them to automatically predict the performance characteristics of different resource types.

8 Conclusion

In this paper we motivated and presented the design of Conductor, a system that assists cloud customers in choosing the right set of resources to use when running cloud computations. Conductor takes the burden of manual choice and optimization away from the customer, by automating the selection process and providing mechanisms to utilize different services seamlessly. This automation and flexibility allows the customer to state high level goals about price or performance, rather than having to make low level resource selection decisions.

Conductor requires users to specify simple high level goals, and a small amount of information about the MapReduce computation, and then uses optimization tools to determine an execution plan. This execution plan is deployed, and then adapted, if any of the information used in computing the plan changes at runtime. Our evaluation shows that Conductor is able to find and deploy non-obvious execution plans, while incurring only a modest overhead.

Conductor is an important first step in automating cloud resource selection, but much work remains in generalizing the set of supported applications, increasing the adaptivity to changing conditions, and providing powerful abstractions for computation that cover a wide range of services.

References

[1] CHOHAN, N., CASTILLO, C., SPREITZER, M., STEINDER, M., TANTAWI, A., AND KRINTZ, C. See Spot Run: Using spot instances for MapReduce workflows. In *2nd USENIX Workshop on Hot Topics in Cloud Computing* (2010).

[2] DEAN, J., AND GHEMAWAT, S. Mapreduce: simplified data processing on large clusters. In *OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation* (Berkeley, CA, USA, 2004), USENIX Association.

[3] DUSSEAU, A. C., ARPACI, R. H., AND CULLER, D. E. Effective distributed scheduling of parallel workloads. In *SIGMETRICS '96: Proceedings of the 1996 ACM SIGMETRICS international conference on Measurement and modeling of computer systems* (1996).

[4] GANAPATHI, A. S. *Predicting and Optimizing System Utilization and Performance via Statistical Machine Learning*. PhD thesis, EECS Department, University of California, Berkeley, 2009.

[5] HAJJAT, M., SUN, X., SUNG, Y.-W. E., MALTZ, D., RAO, S., SRIPANIDKULCHAI, K., AND TAWARMALANI, M. Cloudward bound: planning for beneficial migration of enterprise applications to the cloud. In *Proceedings of the ACM SIGCOMM 2010 conference on SIGCOMM* (New York, NY, USA, 2010), SIGCOMM '10, ACM.

[6] HINDMAN, B., KONWINSKI, A., ZAHARIA, M., GHODSI, A., JOSEPH, A. D., KATZ, R., SHENKER, S., AND STOICA, I. Mesos: a platform for fine-grained resource sharing in the data center. In *Proceedings of the 8th USENIX conference on Networked systems design and implementation* (Berkeley, CA, USA, 2011), NSDI'11, USENIX Association.

[7] ISARD, M., PRABHAKARAN, V., CURREY, J., WIEDER, U., TALWAR, K., AND GOLDBERG, A. Quincy: fair scheduling for distributed computing clusters. In *SOSP '09: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles* (2009).

[8] KEETON, K., KELLY, T., MERCHANT, A., SANTOS, C., WIENER, J., ZHU, X., AND BEYER, D. Don't settle for less than the best: Use optimization to make decisions. In *HOTOS'07: Proceedings of the 11th USENIX workshop on Hot topics in operating systems* (2007).

[9] KO, S. Y., HOQUE, I., CHO, B., AND GUPTA, I. Making cloud intermediate data fault-tolerant. In *Proceedings of the 1st ACM symposium on Cloud computing* (New York, NY, USA, 2010), SoCC '10, ACM.

[10] KRAWCZYK, S., AND BUBENDORFER, K. Grid resource allocation: allocation mechanisms and utilisation patterns. In *AusGrid '08: Proceedings of the sixth Australasian workshop on Grid computing and e-research* (Darlinghurst, Australia, Australia, 2008), Australian Computer Society, Inc.

[11] LIU, H. Cutting mapreduce cost with spot market. In *3rd USENIX Workshop on Hot Topics in Cloud Computing* (2011).

[12] MUTHITACHAROEN, A., MORRIS, R., GIL, T. M., AND CHEN, B. Ivy: a read/write peer-to-peer file system. *SIGOPS Operating Systems Review 36*, SI (2002).

[13] OLSTON, C., REED, B., SRIVASTAVA, U., KUMAR, R., AND TOMKINS, A. Pig latin: a not-so-foreign language for data processing. In *SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data* (2008).

[14] QURESHI, A., WEBER, R., BALAKRISHNAN, H., GUTTAG, J., AND MAGGS, B. Cutting the electric bill for internet-scale systems. In *SIGCOMM '09: Proceedings of the ACM SIGCOMM 2009 conference on Data communication* (2009).

[15] RAMAN, R., LIVNY, M., AND SOLOMON, M. Matchmaking: Distributed resource management for high throughput computing. In *HPDC '98: Proceedings of the 7th IEEE International Symposium on High Performance Distributed Computing* (Washington, DC, USA, 1998), IEEE Computer Society, p. 140.

[16] WIEDER, A., BHATOTIA, P., POST, A., AND RODRIGUES, R. Conductor: orchestrating the clouds. In *Proceedings of the 4th International Workshop on Large Scale Distributed Systems and Middleware (LADIS 2010)*.

[17] WIEDER, A., BHATOTIA, P., POST, A., AND RODRIGUES, R. Brief Announcement: Modelling MapReduce for Optimal Execution in the Cloud. In *PODC '10: Proceedings of the 29th ACM symposium on Principles of distributed computing* (2010).

[18] YIN, Q., SCHÜPBACH, A., CAPPOS, J., BAUMANN, A., AND ROSCOE, T. Rhizoma: a runtime for self-deploying, self-managing overlays. In *Middleware '09: Proceedings of the 10th ACM/IFIP/USENIX International Conference on Middleware* (2009).

[19] ZAHARIA, M., BORTHAKUR, D., SEN SARMA, J., ELMELEEGY, K., SHENKER, S., AND STOICA, I. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In *Proceedings of the 5th European conference on Computer systems* (New York, NY, USA, 2010), EuroSys '10, ACM.

[20] ZAHARIA, M., KONWINSKI, A., JOSEPH, A. D., KATZ, R., AND STOICA, I. Improving mapreduce performance in heterogeneous environments. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation* (Berkeley, CA, USA, 2008), OSDI'08, USENIX Association.

Fitting Square Pegs Through Round Pipes

Unordered Delivery Wire-Compatible with TCP and TLS

Michael F. Nowlan[†] Nabin Tiwari[*] Janardhan Iyengar[*] Syed Obaid Amin[†] Bryan Ford[†]

Abstract

Internet applications increasingly employ TCP not as a *stream abstraction*, but as a *substrate* for application-level transports, a use that converts TCP's in-order semantics from a convenience blessing to a performance curse. As Internet evolution makes TCP's use as a substrate likely to grow, we offer *Minion*, an architecture for backward-compatible out-of-order delivery atop TCP and TLS. Small OS API extensions allow applications to manage TCP's send buffer and to receive TCP segments out-of-order. Atop these extensions, Minion builds application-level protocols offering true unordered datagram delivery, within streams preserving strict wire-compatibility with unsecured or TLS-secured TCP connections. Minion's protocols can run on unmodified TCP stacks, but benefit incrementally when either endpoint is upgraded, for a backward-compatible deployment path. Experiments suggest that Minion can noticeably improve performance of applications such as conferencing, virtual private networking, and web browsing, while incurring minimal CPU or bandwidth costs.

1 Introduction

TCP [46] was originally designed to offer applications a convenient, high-level communication abstraction with semantics emulating Unix file I/O or pipes. As the Internet has evolved, however, TCP's original role of offering an *abstraction* has gradually been supplanted with a new role of providing a *substrate* for transport-like, application-level protocols such as SSL/TLS [17], ØMQ [3], SPDY [2], and WebSockets [52]. In this new *substrate* role, TCP's in-order delivery offers little value since application libraries are equally capable of implementing convenient abstractions. TCP's strict in-order delivery, however, prevents applications from controlling the *framing* of their communications [14, 19], and incurs a "latency tax" on content whose delivery must wait for the retransmission of a single lost TCP segment.

Due to the difficulty of deploying new transports today [20, 36, 41], applications rarely utilize new out-of-order transports such as SCTP [45] and DCCP [28]. UDP [37] is a popular substrate, but is still not universally supported in the Internet, leading even delay-sensitive applications such as the Skype telephony system [7] and Microsoft's DirectAccess VPN [16], to fall back on TCP despite its drawbacks.

Recognizing that TCP's use as a substrate is likely to continue and expand, we introduce *Minion*, a novel architecture for efficient but backward-compatible unordered delivery in TCP. Minion consists of u*TCP*, a small OS extension adding basic unordered delivery primitives to TCP, and two application-level protocols implementing datagram-oriented delivery services that function on either uTCP or unmodified TCP stacks.

uTCP addresses delays caused by TCP's send and receive buffering. On the send side, uTCP gives the application a controlled ability to insert data out-of-order into TCP's send queue, allowing fresh high-priority data to bypass previously-queued low-priority data, for example. On the receive side, uTCP enables the application to receive out-of-order TCP segments immediately, without delaying their delivery until retransmissions fill prior holes. Designed for simplicity and deployability, these extensions add less than 600 lines to Linux's TCP stack.

Minion's application-level protocols, u*COBS* and u*TLS*, build general datagram delivery services atop uTCP or TCP. Key challenges these protocols address are: (a) TCP offers no reliable out-of-band framing to delimit datagrams in a TCP stream; (b) uTCP cannot add out-of-band framing without changing TCP's wire protocol; and (c) common in-band TCP framing methods assume in-order processing. To make datagrams *self-delimiting* in a TCP stream, uCOBS leverages *Consistent Overhead Byte Stuffing* (COBS) [12] to encode application datagrams with at most 0.4% expansion, while reserving a single byte value to delimit encoded datagrams.

Minion adapts the stream-oriented TLS [17] into a secure datagram delivery service atop uTCP or TCP. To avoid changing the TLS wire protocol, the uTLS receiver heuristically "guesses" TLS record boundaries in stream fragments received out-of-order, then leverages TLS's cryptographic MAC to confirm these guesses reliably. By preserving strict wire-compatibility with TLS, uTLS enables unordered delivery within streams indistinguishable in the network from HTTPS [39], for example.

Experiments with a prototype on Linux show several benefits for applications using TCP. Minion can reduce application-perceived jitter of Voice-over-IP (VoIP) streams atop TCP, and increase perceptible-quality metrics [32]. Virtual private networks (VPNs) that tunnel IP traffic over SSL/TLS, such as OpenVPN [34] or Di-

[*]Franklin and Marshall College
[†]Yale University

rectAccess [16], can double the throughput of some tunneled TCP connections, by employing uTCP to prioritize and expedite tunneled ACKs. Web transports can cut the time before a page begins to appear by up to half, achieving the latency benefits of multistreaming transports [19, 31] while preserving the TCP substrate. Use of uCOBS can incur up to $5\times$ CPU load with respect to raw TCP, due to COBS encoding, but for secure connections, uTLS incurs less than 7% CPU overhead (and no bandwidth overhead) atop the baseline cost of TLS 1.1.

This paper's primary contributions are: (a) the first wire-compatible TCP extension we are aware of offering true out-of-order delivery; (b) an API allowing applications to prioritize TCP's send queue; (c) a novel use of COBS [12] for out-of-order framing atop TCP; (d) an existence proof that out-of-order datagram delivery is achievable from the unmodified, stream-based TLS wire protocol; (e) a prototype and experiments demonstrating Minion's practicality and performance benefits.

Section 2 motivates Minion by discussing the evolution of TCP's role in the Internet. Section 3 introduces Minion's high-level architecture, and Section 4 describes its uTCP extensions. Section 5 presents uCOBS for non-secure datagram delivery, and Section 6 details uTLS, a secure analog. Section 7 discusses the current Minion prototype, and Section 8 evaluates its performance experimentally. Section 9 summarizes related work, and Section 10 concludes.

2 Motivating Minion

This section describes how TCP's role in the network has evolved from a communication *abstraction* to a communication *substrate*, why its in-order delivery model makes TCP a poor substrate, and why other OS-level transports have failed to replace TCP in this role.

2.1 Rise of Application-Level Transports

The transport layer's traditional role in a network stack is to build high-level communication abstractions convenient to applications, atop the network layer's basic packet delivery service. TCP's reliable, stream-oriented design [46] exemplified this principle, by offering an inter-host communication abstraction modeled on Unix pipes, which were the standard *intra-host* communication abstraction at the time of TCP's design. The Unix tradition of implementing TCP in the OS kernel offered further convenience, allowing much application code to ignore the difference between an open disk file, an intra-host pipe, or an inter-host TCP socket.

Instead of building directly atop traditional OS-level transports such as TCP or UDP, however, today's applications frequently introduce additional transport-like protocol layers at user-level, typically implemented via application-linked libraries. Examples include the ubiquitous SSL/TLS [17], media transports such as RTP [44], and experimental multi-streaming transports such as SST [19], SPDY [2], and ØMQ [3]. Applications increasingly use HTTP or HTTPS over TCP as a substrate [36]; this is also illustrated by the W3C's WebSocket interface [52], which offers general bidirectional communication between browser-based applications and Web servers atop HTTP and HTTPS.

In this increasingly common design pattern, the "transport layer" as a whole has in effect become a stack of protocols straddling the OS/application boundary. Figure 1 illustrates one example stack, representing Google's experimental Chromium browser, which inserts SPDY for multi-streaming and TLS for security at application level, atop the OS-level TCP.

One can debate whether a given application-level protocol fits some definition of "transport" functionality. The important point, however, is that today's applications no longer need, or expect, the underlying OS to provide "convenient" communication abstractions: the application simply links in libraries, frameworks, or middleware offering the abstractions it desires. What today's applications need from the OS is not convenience, but *an efficient substrate* atop which application-level libraries can build the desired abstractions.

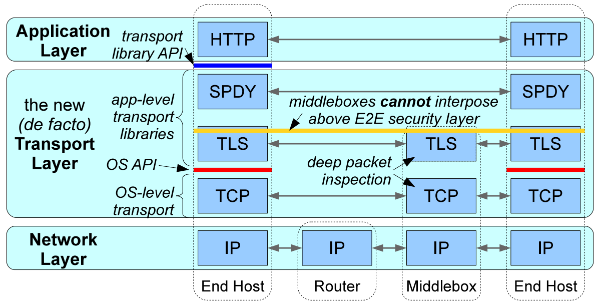


Figure 1: Today's "*de facto* transport layer" is effectively split between OS and application code.

2.2 TCP's Latency Tax

While TCP has proven to be a popular substrate for application-level transports, using TCP in this role converts its delivery model from a blessing into a curse. Application-level transports are just as capable as the kernel of sequencing and reassembling packets into a logical data unit or "frame" [14]. By delaying any segment's delivery to the application until all prior segments are received and delivered, however, TCP imposes a "latency tax" on all segments arriving within one round-trip time (RTT) after any single lost segment.

This latency tax is a fundamental byproduct of TCP's in-order delivery model, and is irreducible, in that an application-level transport cannot "claw back" the time a potentially useful segment has wasted in TCP's buffers.

The best the application can do is simply to *expect* higher latencies to be common. A conferencing application can use a longer jitter buffer, for example, at the cost of increasing user-perceptible lag. Network hardware advances are unlikely to address this issue, since TCP's latency tax depends on RTT, which is lower-bounded by the speed of light for long-distance communications.

2.3 Alternative OS-level Transports

All standardized OS-level transports since TCP, including UDP [37], RDP [51], DCCP [28], and SCTP [45], support out-of-order delivery. The Internet's evolution has created strong barriers against the widespread deployment of new transports other than the original TCP and UDP, however. These barriers are detailed elsewhere [20,36,41], but we summarize two key issues here.

First, adding or enhancing a "native" transport built atop IP involves modifying popular OSes, effectively increasing the bar for widespread deployment and making it more difficult to evolve transport functionality below the red line representing the OS API in Figure 1. Second, the Internet's original "dumb network" design, in which routers that "see" only up to the IP layer, has evolved into a "smart network" in which pervasive middleboxes perform deep packet inspection and interposition in transport and higher layers. Firewalls tend to block "anything unfamiliar" for security reasons, and Network Address Translators (NATs) rewrite the port number in the transport header, making both incapable of allowing traffic from a new transport without explicit support for that transport. Any packet content not protected by end-to-end security such as TLS—the yellow line in Figure 1—has become "fair game" for middleboxes to inspect and interpose on [38], making it more difficult to evolve transport functionality anywhere below that line.

2.4 Why Not UDP?

As the only widely-supported transport with out-of-order delivery, UDP offers a natural substrate for application-level transports. Even applications otherwise well-suited to UDP's delivery model often favor TCP as a substrate, however. A recent study found over 70% of streaming media using TCP [23], and even latency-sensitive conferencing applications such as Skype often use TCP [7].

Network middleboxes support UDP widely but not *universally*. For this reason, latency-sensitive applications seeking maximal connectivity "in the wild" often fall back to TCP when UDP connectivity fails. Skype [7] and Microsoft's DirectAccess VPN [16], for example, support UDP but can masquerade as HTTP or HTTPS streams atop TCP when required for connectivity.

TCP can offer performance advantages over UDP as well. For applications requiring congestion control, an OS-level implementation in TCP may be more timing-

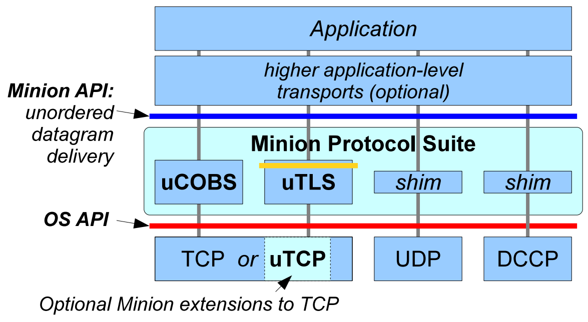

Application
higher application-level transports (optional)
Minion API:
unordered datagram delivery
Minion Protocol Suite
uCOBS
uTLS
shim
shim
OS API
TCP or uTCP
UDP
DCCP
Optional Minion extensions to TCP

Figure 2: Minion architecture

accurate than an application-level implementation in a UDP-based protocol, because the OS kernel can avoid the timing artifacts of system calls and process scheduling [54]. Hardware TCP offload engines can optimize common-case efficiency in end hosts [30], and performance enhancing proxies can optimize TCP throughput across diverse networks [11, 13]. Since middleboxes can track TCP's state machine, they impose much longer idle timeouts on open TCP connections—nominally two hours [22]—whereas UDP-based applications must send keepalives every two minutes to keep an idle connection open [5], draining power on mobile devices.

For applications, TCP versus UDP represents an "all-or-nothing" choice on the spectrum of services applications need. Applications desiring some but not all of TCP's services, such as congestion control but unordered delivery, must reimplement and tune all other services atop UDP or suffer TCP's performance penalties.

Without dismissing UDP's usefulness as a truly "least-common-denominator" substrate, we believe the factors above suggest that TCP will also remain a popular substrate—even for latency-sensitive applications that can benefit from out-of-order delivery—and that a deployable, backward-compatible workaround to TCP's latency tax can significantly benefit such applications.

3 Minion Architecture Overview

Minion is an architecture and protocol suite designed to meet the needs of today's applications for efficient unordered delivery built atop either TCP or UDP. Minion itself offers no high-level abstractions: its goal is to serve applications and higher application-level transports, by acting as a "packhorse" carrying raw datagrams as reliably and efficiently as possible across today's diverse and change-averse Internet.

3.1 All-Terrain Unordered Delivery

Figure 2 illustrates Minion's architecture. Applications and higher application-level transports link in and use Minion in the same way as they already use existing application-level transports such as DTLS [40], the datagram-oriented analog of SSL/TLS [17]. In contrast with DTLS's goal of layering security atop datagram transports such as UDP or DCCP, Minion's goal is to offer efficient datagram delivery atop *any* available OS-level substrate, including TCP.

While many protocols embed datagrams or application-level frames into TCP streams using delimiting schemes, to our knowledge Minion is the first application-level transport that, under suitable conditions, offers true unordered delivery atop TCP. Minion effectively offers relief from TCP's latency tax: the loss of one TCP segment in the network no longer prevents datagrams embedded in subsequent TCP segments from being delivered promptly to the application.

3.2 Minion Architecture Components

Minion consists of several application-level transport protocols, together with a set of optional enhancements to end hosts' OS-level TCP implementations.

Minion's enhanced OS-level TCP stack, which we call *uTCP* ("unordered TCP"), includes sender- and receiver-side API features supporting unordered delivery and prioritization, detailed in Section 4. These enhancements affect only the OS API through which application-level transports such as Minion interact with the TCP stack, and make *no* changes to TCP's wire protocol.

Minion's application-level protocol suite currently consists of uCOBS, which implements unordered datagram delivery atop unmodified TCP or uTCP streams using COBS encoding [12] as described in Section 5; and uTLS, which adapts the traditionally stream-oriented TLS [17] into a secure unordered datagram delivery service atop TCP or uTCP. Minion also adds trivial shim layers atop OS-level datagram transports, such as UDP and DCCP, to give applications a consistent API for unordered delivery across multiple OS-level transports.

Minion currently leaves to the application the decision of *which* protocol to use for a given connection: e.g., uCOBS or uTLS atop TCP/uTCP, or OS-level UDP or DCCP via Minion's shims. We are developing an experimental *negotiation protocol* to explore the protocol configuration space dynamically, optimizing protocol selection and configuration for the application's needs and the network's constraints [21], but we defer this enhancement to future work. Many applications already incorporate simple negotiation schemes, however—e.g., attempting a UDP connection first and falling back to TCP if that fails—and adapting these mechanisms to engage Minion's protocols according to application-defined preferences and decision criteria should be straightforward.

3.3 Compatibility and Deployability

Minion addresses the key barriers to transport evolution, outlined in Section 2.3, by creating a backward-compatible, incrementally deployable substrate for new application-layer transports desiring unordered delivery. Minion's deployability rests on the fact that it can, when necessary, avoid relying on changes either "below the red line" in the end hosts (the OS API in Figure 1), or "below the yellow line" in the network (the end-to-end security layer in Figure 1).

While Minion's uCOBS and uTLS protocols offer maximum performance benefits from out-of-order delivery when both endpoints include OS support for Minion's uTCP enhancements, uCOBS and uTLS still function and interoperate correctly even if neither endpoint supports uTCP, and the application need not know or care whether the underlying OS supports uTCP. If only one endpoint OS supports uTCP, Minion still offers incremental performance benefits, since uTCP's sender-side and receiver-side enhancements are independent. A uCOBS or uTLS connection atop a mixed TCP/uTCP endpoint-pair benefits from uTCP's sender-side enhancements for datagrams sent by the uTCP endpoint, and the connection benefits from uTCP's receiver-side enhancements for datagrams arriving at the uTCP host.

Addressing the challenge of network-compatibility with middleboxes that filter new OS-level transports and sometimes UDP, Minion offers application-level transports a continuum of substrates representing different tradeoffs between suitability to the application's needs and compatibility with the network. An application can use unordered OS-level transports such as UDP, DCCP [28], or SCTP [45], for paths on which they operate, but Minion offers an unordered delivery alternative usable even when TCP is the only viable choice.

A final issue is compatibility with existing applications. Since most of Minion operates at application-level, applications must be changed to use the Minion API. A pair of application endpoints may also need to negotiate whether to use Minion, or to run directly atop OS-level transports for compatibility with earlier versions of the application. This challenge is comparable to the cost of adding TLS or DTLS support to an application, and the popularity of application-level transports such as TLS suggests that these costs are surmountable. Minion's application-level functionality might eventually be merged into existing or future application-level transports and communication frameworks, making its benefits available with few or no application changes.

4 uTCP: Unordered TCP

Minion enhances the OS's TCP stack with API enhancements supporting unordered delivery in both TCP's send and receive paths, enabling applications to reduce transmission latency at both the sender- and receiver-side end hosts when both endpoints support uTCP. Since uTCP makes no changes to TCP's wire protocol, two endpoints need not "agree" on whether to use uTCP: one endpoint gains latency benefits from uTCP even if the other endpoint does not support it. Further, an OS may choose independently whether to support the sender- and receiver-side enhancements, and when available, applications can activate them independently.

In this spirit of Section 2, uTCP does *not* seek to offer "convenient" or "clean" unordered delivery abstractions directly at the OS API. Instead, uTCP's design is motivated by the goals of maintaining exact compatibility with TCP's existing wire-visible protocol and behavior, and facilitating deployability by minimizing the extent and complexity of changes to the OS's TCP stack. The design presented here is only one of many viable approaches, with different tradeoffs, to supporting unordered delivery in TCP. Section 4.3 briefly outlines a few such alternatives.

We describe uTCP's API enhancements in terms of the BSD sockets API, although uTCP's design contains nothing inherently specific to this API.

4.1 Receiver-Side Enhancements

uTCP adds one new socket option affecting TCP's receive path, enabling applications to request immediate delivery of TCP segments received out of order. An application opens a TCP stream the usual way, via `connect()` or `accept()`, and may use this stream for conventional in-order communication before enabling uTCP. Once the application is ready to receive out-of-order data, it enables the new option `SO_UNORDERED` via `setsockopt()`, which changes TCP's receive-side behavior in two ways.

First, whereas a conventional TCP stack delivers received data to the application only when prior gaps in the TCP sequence space are filled, the uTCP receiver makes data segments available to the application immediately upon receipt, skipping TCP's usual reordering queue. The application obtains this data via `read()` as usual, but the first data byte returned by a `read()` call may no longer be the one logically following the last byte returned by the prior `read()` call, in the byte stream transmitted by the sender. The data the uTCP stack delivers to the application in successive `read()` calls may skip forward and backward in the transmitted byte stream, and uTCP may even deliver portions of the transmitted stream multiple times. uTCP guarantees only that the data returned by one `read()` call corresponds to *some* contiguous sequence of bytes in the sender's transmitted stream, and that barring connection failure, uTCP will *eventually* deliver every byte of the transmitted stream at least once.

Second, when servicing an application's `read()` call, the uTCP receiver prepends a short header to the returned data, indicating the logical offset of the first returned byte in the sender's original byte stream. The uTCP stack computes this logical offset simply by subtracting the Initial Sequence Number (ISN) of the received stream from the TCP sequence number of the segment being delivered. Using this metadata, the application can piece together data segments from successive `read()` calls into longer contiguous *fragments* of the transmitted byte stream.

Figure 3 illustrates uTCP's receive-side behavior, in a simple scenario where three TCP segments arrive in succession: first an in-order segment, then an out-of-order segment, and finally a segment filling the gap between the first two. With uTCP, the application receives each segment as soon as it arrives, along with the sequence number information it needs to reconstruct a complete internal view of whichever fragments of the TCP stream have arrived.

The uTCP receiver retains in its receive buffer the TCP headers of segments received and delivered out-of-order, until its cumulative acknowledgment point moves past these segments, and generates acknowledgments and selective acknowledgments (SACKs) exactly as TCP normally would. The uTCP receiver does not increase its advertised receive window when it delivers data to the application out-of-order, so the advertised window tracks the cumulative in-order delivery point exactly as in TCP. In this fashion, uTCP maintains wire-visible behavior identical to TCP while delivering segments to the application out-of-order.

The uTCP receive path assumes the sender may be an unmodified TCP, and TCP's stream-oriented semantics allow the sending TCP to segment the sending application's stream at arbitrary points—independent of the boundaries of the sending application's `write()` calls, for example. Further, network middleboxes may silently *re-segment* TCP streams, making segment boundaries observed at the receiver differ from the sender's original transmissions [24]. An application using uTCP, therefore, must not assume anything about received segment boundaries. This is a key technical challenge to using uTCP reliably, and addressing this challenge is one function of uCOBS and uTLS, described later.

4.2 Sender-Side Enhancements

While uTCP's receiver-side enhancements address the "latency tax" on segments waiting in TCP's reordering buffer, TCP's sender-side queue can also introduce latency, as segments the application has already written to a TCP socket—and hence "committed" to the network—wait until TCP's flow and congestion control allow their transmission. Many applications can benefit from the ability to "late-bind" their decision on *what* to send until the last possible moment. An application-level multistreaming transport with prioritization, such as SST [19]

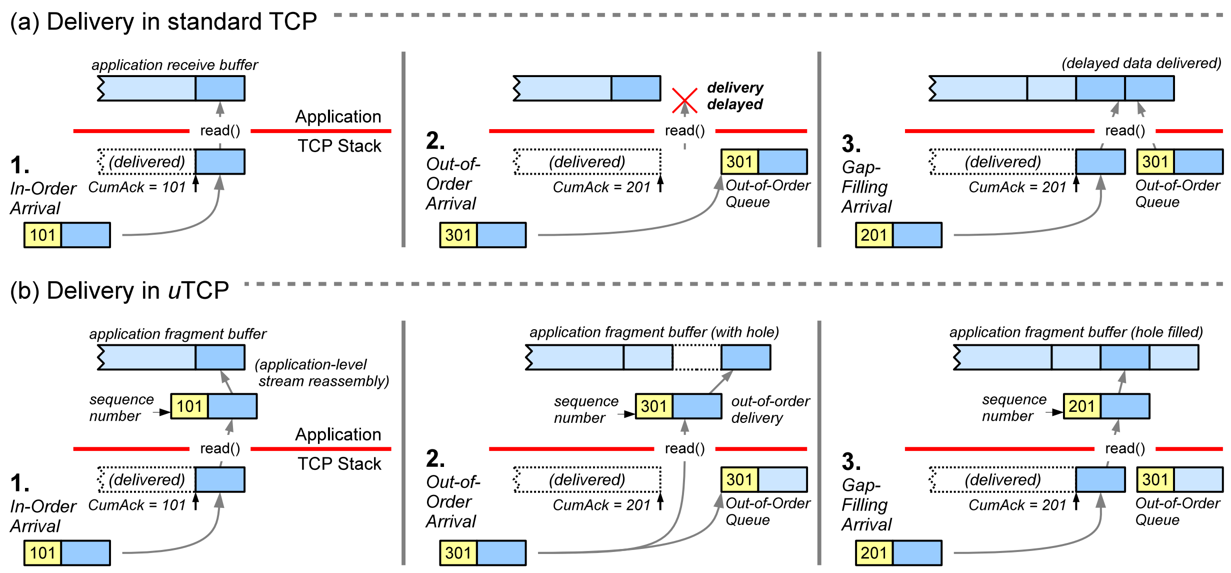


Figure 3: Delivery behavior of (a) standard TCP, and (b) uTCP, upon receipt of in-order and out-of-order segments.

or SPDY [2], would prefer high-priority packets not to get "stuck" behind low-priority packets in TCP's send queue. In applications such as games and remote-access protocols, where the receiver typically desires *only the freshest* in a stream of real-time status updates, the sender would prefer that new updates "squash" any prior updates still in TCP's send queue and not yet transmitted.

The Congestion Manager architecture [6] addressed this desire to "late-bind" the application's transmission decisions, by introducing an upcall-based API in which the OS performs no send buffering, but instead signals the application whenever the application is permitted to send. Upcalls represent a major change to conventional sockets APIs, however, and introduce issues such as how the OS should handle an application that fails to service an upcall promptly, leaving its allocated transmission time-slot unfilled yet unavailable to competing applications waiting to send.

In the spirit of maximizing deployability, uTCP adopts a more limited but less invasive design, by retaining TCP's send buffer but giving applications some control over it. After enabling uTCP's new socket option `SO_UNORDEREDSEND`, the OS expects any subsequent `write()` to that socket to include a short header, containing metadata that uTCP reads and strips before placing the remaining data on TCP's send buffer. The uTCP header contains an integer *tag* and a set of optional flags controlling uTCP's send-side behavior.

By default, uTCP interprets tags as priority levels. Instead of unconditionally placing the newly-written data at the tail of the send queue as TCP normally would, uTCP *inserts* the newly-written data into the send queue just *before* any lower-priority data in the send queue and not yet transmitted. The application thus avoids higher-priority packets being delayed by lower-priority packets enqueued earlier, while the OS avoids the complexity and security challenges of an upcall API.

For strict TCP wire-compatibility, uTCP never inserts new data into the send queue ahead of any previously-written data that has ever been transmitted in whole or in part: e.g., ahead of data from a prior application write already partly transmitted and awaiting acknowledgment. If an application writes a large low-priority buffer, then writes higher-priority data after transmission of the low-priority data has begun, uTCP inserts the high-priority data after the entire low-priority write and never in the middle. This constraint enables the application to control the boundaries on which send buffer reordering is permitted, independent of the current MTU and TCP segmentation behavior.

A simple uTCP refinement, which we intend to explore in future work, is to include a *squash* flag in the metadata header the application prepends to each write. If set, while inserting the newly-written data in tag-priority order, uTCP would also remove and discard any data previously written with exactly the same tag, that has not yet been transmitted in whole or in part. This refinement would enable update-oriented applications such as games to avoid the bandwidth cost transmitting old updates superseded by newer data.

4.3 Design Alternatives

uTCP pursues a conservative point in a large design space, and many alternatives present interesting trade-offs. Some alternatives include: disabling TCP congestion control at the sender; assigning TCP sequence numbers at application write time instead of the time a segment is first transmitted; sending new data in retransmitted segments; modifying the receiver to acknowledge unreceived sequence space gaps for unreliable service; increasing the receive window to account for out-of-order segments; and delivering data to the application exactly-once instead of at-least-once. For space reasons we discuss these tradeoffs in more detail elsewhere [26]. A common theme, however, is that most of these design alternatives change TCP's behavior in wire-visible ways, which can trigger various unpredictable middlebox behaviors [24], making connectivity less reliable.

5 uCOBS: Simple Datagrams on TCP

Since uTCP's design attempts to minimize OS changes, its unordered delivery primitives do not directly offer applications a convenient, general-purpose datagram substrate. Minion's uCOBS protocol bridges this semantic gap, building atop uTCP (or standard TCP) a lightweight datagram delivery service comparable to UDP or DCCP. This first section first introduces the challenge of delimiting datagrams, then presents uCOBS' solution and discusses alternatives.

5.1 Self-Delimiting Datagrams for uTCP

Applications built on datagram substrates such as UDP generally assume the underlying layer preserves datagram boundaries. If the network fragments a large UDP datagram, the receiving host reassembles it before delivery to the application, and a correct UDP never merges multiple datagrams, or datagram fragments, into one delivery to the receiving application. TCP's stream-oriented semantics do not preserve any application-relevant frame boundaries within a stream, however. Both the TCP sender and network middleboxes can and do coalesce TCP segments or re-segment TCP streams in unpredictable ways [24]. Conventional TCP applications, which send and receive TCP data in-order, commonly address this issue by delimiting application-level frames with some length-value encoding, enabling the receiver to locate the next frame in the stream from the previous frame's position and header content.

Since uTCP's receive path effectively just bypasses TCP's reordering buffer, delivering received segments to the application as they arrive, a stream fragment received out-of-order from uTCP may begin at any byte offset in the stream, and not at a frame boundary meaningful to the application. Since the receiver is by definition missing some data sent prior to this out-of-order segment, it cannot rely on preceding stream content to compute the next frame's position.

Reliable use of uTCP, therefore, requires that frames embedded in the TCP stream be *self-delimiting*: recognizable without knowledge of preceding or following data. A simple solution is to make frames fixed-length, so the receiver can compute the start of the next frame from the stream offset uTCP provides with out-of-order segments. uCOBS is intended to offer a general-purpose datagram substrate, however, and many applications require support for variable-length frames.

If the application-level frames happen to be encoded so as never to include some "reserved" byte value, such as zero, then we could use that byte reserved value to delimit frames within uTCP streams. Since we wish uCOBS to support general-purpose delivery of datagrams of variable length containing arbitrary byte values, however, uCOBS must explicitly (re-)encode the application's datagrams in order to reserve some byte value to serve as a delimiter.

Any scheme that encodes arbitrary byte streams into strings utilizing fewer than 256 symbols will serve this purpose, such as the ubiquitous *base64* scheme, which encodes byte streams into strings utilizing only 64 ASCII symbols plus whitespace. Since base64 encodes three bytes into four ASCII symbols, however, it expands encoded streams by a factor of $4/3$, incurring a 33% bandwidth overhead. Since uCOBS needs to reserve only *one* byte value for delimiters, and not the large set of byte values considered "unsafe" in E-mail or other text-based message formats, base64 encoding is unnecessarily conservative for uCOBS' purposes.

5.2 Operation of uCOBS

To encode application datagrams efficiently, uCOBS employs *consistent-overhead byte stuffing*, or COBS [12]. COBS is analogous to base64, except that it encodes byte streams to reserve only *one* distinguished byte value (e.g., zero), and utilizes the remaining 255 byte values in the encoding. COBS could in effect be termed "base255" encoding. By reserving only one byte value, COBS incurs an expansion ratio of at most 255/254, or 0.4% bandwidth overhead.

Transmission: When an application sends a datagram, uCOBS first COBS-encodes the datagram to remove all zero bytes. uCOBS then prepends a zero byte to the encoded datagram, appends a second zero byte to the end, and writes the encoded and delimited datagram to the TCP socket. Since this sender-side encoding and transmission process operates entirely at application level within uCOBS, and does not rely on any OS-level extensions on the sending host, uCOBS operates even if the sender-side OS does not support uTCP.

The application can assign priorities to datagrams it submits to uCOBS, however, and if the sender's OS does support the uTCP extensions in Section 4.2, uCOBS passes these priorities to the uTCP sender, enabling higher-priority datagrams to pass lower-priority datagrams already enqueued. Since uTCP respects application `write()` boundaries while reordering the send queue, uCOBS preserves its delimiting invariant simply

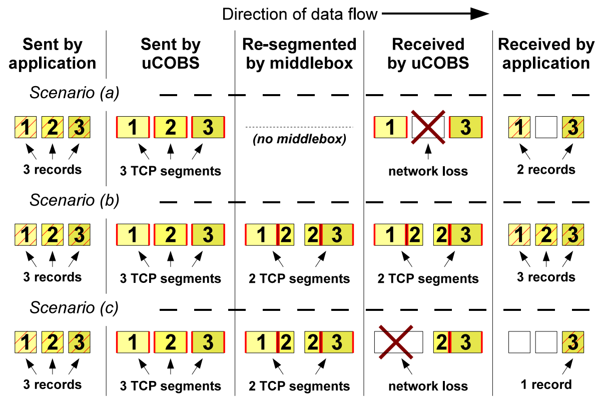


Figure 4: An example illustrating a uCOBS transfer

by writing each encoded datagram—with the leading and trailing zero bytes—in a single write.

Reception: At stream creation time, uCOBS enables uTCP's receive-side extensions if available. If the receive-side OS does not support uTCP, then uCOBS simply falls back on the standard TCP API, receiving, COBS-decoding, and delivering datagrams to the application in the order they appear in the TCP sequence space. (This may not be the application's original send order if the send-side OS supports uTCP.)

If the receive-side OS supports uTCP, then uCOBS receives segments from uTCP in whatever order they arrive, then fits them together using the metadata in uTCP's headers to form contiguous fragments of the TCP stream. The arrival of a TCP segment can cause uCOBS to create a new fragment, expand an existing fragment at the beginning or end, or "fill a hole" between two fragments and merge them into one. The portion of the TCP stream before the receiver's cumulative-acknowledgment point, containing no sequence holes, uCOBS treats as one large "fragment." uCOBS scans the content of any new, expanded, or merged fragment for properly delimited records not yet delivered to the application. uCOBS identifies a record by the presence of two marker bytes surrounding a contiguous sequence of bytes containing no markers or holes. Once uCOBS identifies a new record, it strips the delimiting markers, decodes the COBS-encoded content to obtain the original record data, and delivers the record to the application.

5.3 Why Two Markers Per Datagram?

For correctness alone, uCOBS need only prepend *or* append a marker byte to each record—not both—but such a design could reduce performance by eliminating opportunities for out-of-order delivery. Consider Scenario (a) in Figure 4, in which an application sends three records. uCOBS encodes these records and sends them via three `write()` calls, which TCP in turn sends in three separate TCP segments. In this scenario, no middleboxes re-segment the TCP stream in the network, but the middle segment is lost. If the uCOBS sender were only to *prepend* a marker at the start of each record, the uCOBS receiver could not deliver record 1 immediately on receipt, since it cannot tell if record 1 extends into the following "hole" in sequence space. Similarly, if the sender were only to *append* a marker at the end of each record, then uCOBS could not deliver segment 3 immediately on receipt, since record 3 might extend backwards into the preceding hole. By adding markers to both ends of each record, uCOBS ensures that the receiver can deliver each record as soon as all of its segments arrive.

These markers enable uCOBS to offer reliable out-of-order delivery even if network middleboxes re-segment the TCP stream. In Scenario (b) in Figure 4, for example, uCOBS sends three records encoded into three TCP segments as above, but a middlebox re-segments them into two longer TCP segments, whose boundary splits record 2 into two parts. If neither of these segments are lost, then the uCOBS receiver can deliver record 1 immediately upon receipt of the first TCP segment, and can deliver records 2 and 3 upon receipt of the second segment. If the first segment is lost as shown in Scenario (c), however, the uCOBS receiver cannot deliver the missing record 1 or the partial record 2, but can still deliver record 3 as soon as the second TCP segment arrives.

6 uTLS: Secure Datagrams on TCP

While uCOBS offers out-of-order delivery wire-compatible up to the TCP level, middleboxes often inspect and manipulate the *content* of TCP streams as well [38]. All unencrypted network traffic today is, *de facto*, "fair game" for middleboxes—and streams exhibiting any "out of the ordinary" middlebox-visible behavior are likely to fail over *some* middleboxes [24]. An application's only way to protect "end-to-end" communication in practice, therefore, is via end-to-end encryption and authentication. But network-layer mechanisms such as IPsec [27] face the same deployment challenges as new secure transports [19], and remain confined to the niche of corporate VPNs. Even VPNs are shifting from IPsec toward HTTPS tunnels [16], the only form of end-to-end encrypted connection almost universally supported on today's Internet. A network administrator or ISP might disable nearly any other port while claiming to offer "Internet access," but would be hard-pressed to disable HTTPS, today's foundation for E-commerce.

We could layer TLS atop uCOBS, but TLS decrypts and delivers data only in-order, negating uTCP's benefit. We could also layer the datagram-oriented DTLS [40] atop uCOBS, but the resulting (DTLS-encrypted *then* COBS-encoded) wire format would be radically different from TLS over TCP, and likely fail to traverse middleboxes expecting TLS, particularly on port 443.

The goal of uTLS, therefore, is to coax out-of-order delivery from the *existing* TCP-oriented TLS wire format, producing an encrypted datagram substrate indistinguishable on the wire from standard TLS connections (except via analysis of "side-channels" such as packet length and timing, which we do not address). Run on port 443, a uTLS stream is indistinguishable from HTTPS—regardless of whether the application actually uses HTTP headers, since the HTTP portion of HTTPS streams are TLS-encrypted anyway. Deployed this way, uTLS effectively offers an end-to-end protected substrate in the "HTTP as the new narrow waist" philosophy [36].

6.1 Design of uTLS

TLS [17] already breaks its communication into *records*, encrypts and authenticates each record, and prepends a header for transmission on the underlying TCP stream. TLS was designed to decrypt records strictly in-order, however, creating three challenges for uTLS:

- **Locating record headers out-of-order.** Since encrypted data may contain any byte sequence, there is no reliable way to differentiate a TLS header from record data in the TCP stream, as COBS encoding provides.
- **Encryption state chaining.** Some TLS ciphersuites chain encryption state across records, making records indecipherable until prior records are processed.
- **Record numbers used in MAC computation.** TLS includes a record number, which increases by 1 for each record, in computing the record's MAC. But the uTLS receiver may not know an out-of-order record's number: holes in TCP sequence space before the record could contain an unknown number of prior records.

To locate records out-of-order, uTLS first scans a received stream fragment for byte sequences that *may* represent the TLS 5-byte header: i.e., containing the correct record type and version, and a plausible length. While this scan may yield false positives, uTLS verifies the inferred header by attempting to decrypt and authenticate the record. If the cryptographic MAC check fails, instead of aborting the connection as TLS normally would, uTLS assumes a false positive and continues scanning.

Since TLS's MAC is designed to prevent resourceful adversaries from constructing a byte sequence the receiver could misinterpret as a record, and it is by definition at least as hard to find such a sequence "accidentally" as to forge one maliciously, TLS security should protect equally well against accidental false positives. One exception is when TLS is using its "null ciphersuite," which performs no packet authentication. With this ciphersuite, normally used only during initial key negotiation, uTLS disables out-of-order delivery to avoid the risk of accepting and delivering false records.

The only obvious solution to the second challenge above is to avoid ciphersuites that chain encryption state across records. Most ciphersuites before TLS 1.1 chain encryption state, unfortunately. Any stream cipher inherently does so, such as the RC4 cipher used in early SSL versions. Most recent ciphersuites use block ciphers in CBC mode. CBC ciphers do not inherently depend on chained encryption state, but do require an Initialization Vector (IV) for each record. Until recently, TLS produced each record's IV implicitly from the prior record's encryption state, making records interdependent.

To fix a security issue, however, TLS 1.1 block ciphers use explicit IVs, which the sender generates independently for each record and prepends to the record's ciphertext. As a side-effect, TLS 1.1 block ciphers support out-of-order decryption. Since TLS supports negotiation of versions and ciphersuites, uTLS simply leverages this process. An application can insist on TLS 1.1 with a block cipher to ensure out-of-order delivery support, or it can permit older ciphersuites to maximize interoperability, at the risk of sacrificing out-of-order delivery.

The third challenge is the implicit "pseudo-header" TLS uses in computing the MAC for each packet. This pseudo-header includes a "sequence number" that TLS increments once per *record*, rather than per *byte* as with TCP sequence numbers. When uTLS identifies a possible TLS record in a TCP fragment received out-of-order, the receiver knows only the byte-oriented TCP stream offset, and not the TLS record number. Since records are variable-length, unreceived holes prior to a record to be authenticated may "hide" a few large records or many smaller records, leaving the receiver uncertain of the correct record number for the MAC check.

To authenticate records out-of-order without modifying the TLS ciphersuite, therefore, uTLS attempts to *predict* the record's likely TLS record number, using heuristics such as the average size of past records, and may try several adjacent record numbers to find one for which the MAC check succeeds. If uTLS fails to find a correct TLS record number, it cannot deliver the record out-of-order, but will still eventually deliver the record in-order.

The current uTLS supports only receiver-side unordered delivery, and not the send-side uTCP enhancements in Section 4.2, because send-side reordering complicates record number prediction. A future enhancement we intend to explore is for uTLS to prepend an explicit record number to application payloads before encryption. Since encryption does not depend on record number, the receiving uTLS can decrypt the record number for use in the MAC check, avoiding the need to predict record numbers and enabling send-side reordering. Since the only wire-protocol change is protected by encryption, the change would be invisible to middleboxes. Preserving end-to-end backward compatibility may require a way to negotiate "under encryption" the use of explicit record numbers, however.

7 Prototype Implementation

This section describes the current Minion prototype, which implements uTCP in the Linux kernel, and implements uCOBS and uTLS in application-linked libraries. The uTCP prototype is Linux-specific, but we expect the API extensions it implements and the application-level libraries to be portable.

The uTCP Receiver in Linux: The uTCP prototype adds about 240 lines and modifies about 50 lines of code in the Linux 2.6.34 kernel, to support the new `SO_UNORDERED` socket option. This extension involved two main changes. First, uTCP modifies the TCP code that delivers segments to the application, to prepend a 5-byte metadata header to the data returned from each `read()` system call. This header consists of a 1-byte flags field and a 4-byte TCP sequence number. One flag bit is currently used, with which uTCP indicates whether it is delivering data in-order or out-of-order. Second, if TCP's in-order queue is empty, uTCP's `read()` path checks and returns data from the out-of-order queue. To minimize kernel changes, segments remain in the out-of-order queue after delivery, so uTCP will eventually deliver the same data again in-order.

The uTCP Sender in Linux: On the send path, uTCP adds about 250 lines of kernel code and modifies about 20 in Linux 2.6.34, supporting a new `SO_UNORDEREDSEND` socket option via two changes.

First, uTCP expects the application to prepend a 5-byte header, containing a 1-byte flags field and a 4-byte tag, to the data passed to each `write()`. The flags are currently unused, and the tag indicates message priority.

Second, uTCP inserts the data from each `write()` into the kernel's send queue in priority order. Linux's TCP send queue is a simple FIFO that packs application data into kernel buffers sized to the TCP connection's Maximum Segment Size (MSS). When inserting application messages non-sequentially, however, uTCP must preserve application message boundaries in the kernel. For simplicity, uTCP allocates kernel buffers (`skbuffs`) so that each message sent via uTCP starts a new `skbuff`, and may span several `skbuffs`, but no `skbuff` contains data from multiple application writes.

Disabling Linux's usual packing of MSS-sized `skbuffs` can affect Linux's congestion control, unfortunately, which counts `skbuffs` sent instead of bytes. Section 8.1 discusses the effects of this Linux-specific issue, which a future version of uTCP will address.

The uCOBS Library: The uCOBS prototype is a user-space library in C, amounting to ~700 lines of code [15]. uCOBS presents simple `cobs_sendmsg()` and `cobs_recvmsg()` interfaces enabling applications to send and receive COBS-encoded datagrams, taking advantage of send-side prioritization and out-of-order reception depending on the presence of send- and receive-side OS support for uTCP, respectively.

A uTLS Prototype Based on OpenSSL: The uTLS prototype builds on OpenSSL 1.0.0 [33], adding ~550 lines of code and modifying ~40 lines [15]. Applications use OpenSSL's normal API to create a TLS connection atop a TCP socket, then set a new uTLS-specific socket option to enable out-of-order, record-oriented delivery on the socket. OpenSSL 1.0.0 unfortunately does not yet support TLS 1.1, the first TLS version that uses explicit Initialization Vectors (IVs), permitting out-of-order decryption. For experimentation, therefore, the uTLS prototype modifies OpenSSL's TLS 1.0 ciphersuite to use explicit IVs as in TLS 1.1. Since this change breaks OpenSSL's interoperability, our prototype is not suitable for deployment. We are currently porting uTLS to the next major OpenSSL release, which supports TLS 1.1.

8 Performance Evaluation

This section evaluates Minion via experiments designed to approximate realistic application scenarios. All experiments were run across three Intel PCs running Linux 2.6.34: between two machines representing end hosts, a third machine interposes on the path and uses dummynet [9] to emulate various network conditions. To minimize well-known TCP delays fairly for both TCP and uTCP, we enabled Linux's "low latency" TCP code path via the `net.ipv4.tcp_low_latency` sysctl, and disabled the Nagle algorithm.

8.1 Bandwidth and CPU Costs

We first explore uTCP's costs, with and without record encoding and extraction via uCOBS and uTLS, for a 30MB bulk transfer on a path with 60ms RTT.

Raw uTCP: uTCP's CPU costs at both the sender and the receiver, without application-level processing, are almost identical to TCP's CPU costs, across a range of loss rates from 0% to 5% (figure omitted for space reasons).

Figure 5 shows bandwidth achieved by raw uTCP and TCP, for different application `write()` sizes. When the message size is a multiple of TCP's Maximum Segment Size (MSS)—at 1448 bytes (1 MSS) and 2896 bytes (2 x MSS)—uTCP's throughput is the same as TCP's.

The disparity elsewhere is due to Linux's congestion control counting `skbuffs` instead of bytes, mentioned earlier in Section 7. We partially address this issue by coalescing data into `skbuffs` where easily possible. More specifically, we coalesce small messages when they fully fit within MSS-sized `skbuffs` at the tail of the sender-side socket buffer. This fix makes uTCP throughput sim-

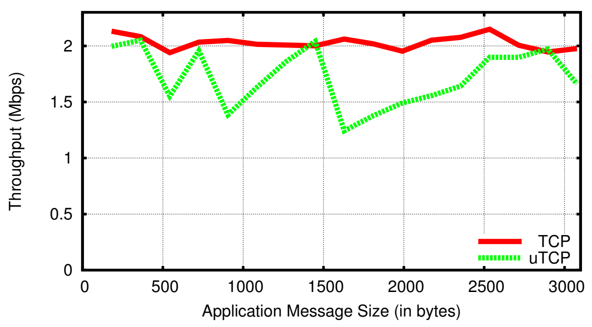


Figure 5: Throughput with different app message sizes.

ilar to TCP's when the MSS is divisible by message size—at 362 bytes ($\frac{1}{4}$ MSS) and 724 bytes ($\frac{1}{2}$ MSS). Future versions of uTCP will fully address this Linux-specific issue with changes either to uTCP or to Linux's congestion control.

Costs with uCOBS/uTLS: To measure these CPU costs, we run a 30MB bulk transfer over a path with a 60ms RTT, for several loss rates.

Figure 6(a) shows CPU costs including application-level encoding/decoding, atop standard TCP ("COBS") and atop uTCP ("uCOBS"), for several loss rates at both sender and receiver. The lighter part of each bar represents user time and the darker part represents kernel time. These results are normalized to the performance of raw TCP, with no application-level encoding or decoding.

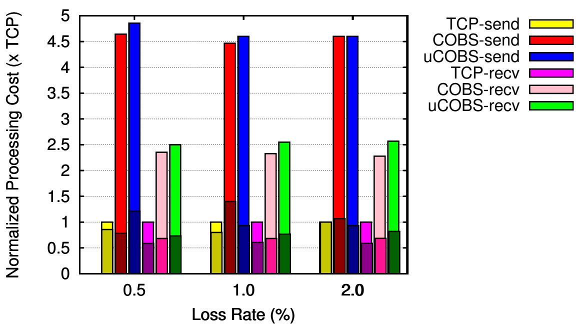
COBS encoding/decoding barely affects kernel CPU use but incurs some application-level CPU cost. This cost is partly due to the encoding itself, and partly because the libraries are not yet well-optimized.

Figure 6(b) shows the CPU costs of uTLS relative to TLS. At the sender, the CPU costs are identical, since there is nothing that uTLS does differently than TLS, and since the CPU cost of using uTCP is practically the same as with TCP. The user-space cost for the uTLS receiver is generally higher than TLS, since the uTLS receiver does more work in processing out-of-order frames than the TLS receiver, but this cost remains within 7% of the TLS receiver's cost.

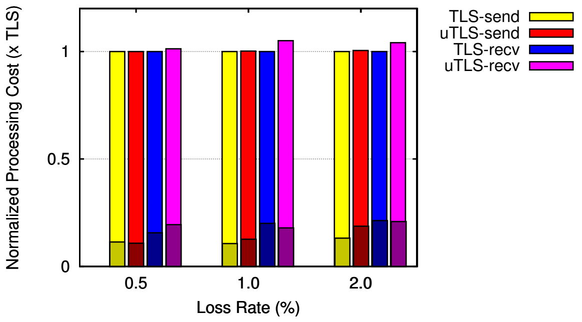
The bandwidth penalty of uCOBS encoding is barely perceptible, under 1%. TLS's bandwidth overhead, up to 10%, is due to TLS headers, IVs, and MACs; uTLS adds no bandwidth overhead beyond standard TLS 1.1.

8.2 Conferencing Applications

We now examine a real-time Voice-over-IP (VoIP) scenario. A test application uses the SPEEX codec [50] to encode a WAV file using ultra-wideband mode (32kHz), for a 256kbps average bit-rate, and transmit voice frames at fixed 20ms intervals. Network bandwidth is 3Mbps and RTT is 60ms, realistic for a home broadband connection. To generate losses more realistically representing



(a) COBS/uCOBS encoding costs



(b) TLS/uTLS costs

Figure 6: CPU costs of using an application with TCP, COBS, and uCOBS at different loss rates.

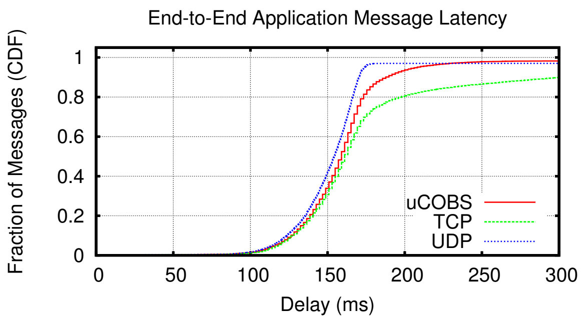


Figure 7: CDF of end-to-end latency in VoIP frames.

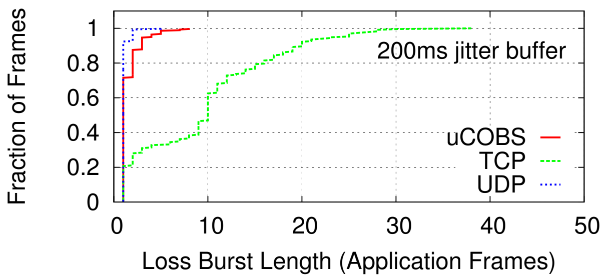


Figure 8: CDF of codec-perceived loss-burst size with TLV encoded frames over TCP, UDP, and uCOBS.

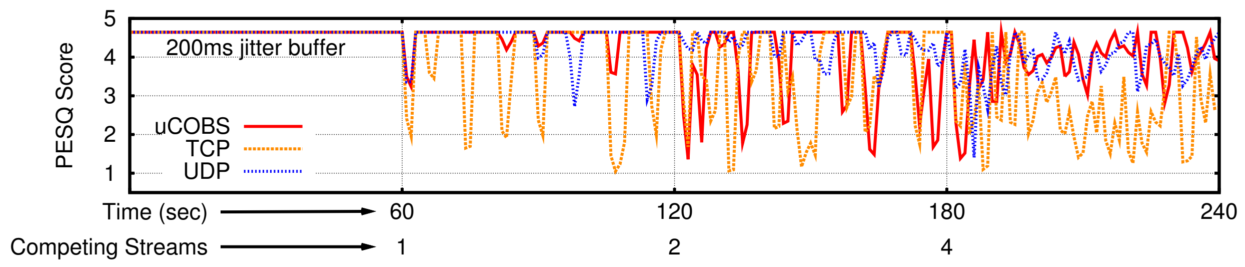


Figure 9: Moving PESQ score of VoIP call under increasing bandwidth competition.

network contention, we run a varying number of competing TCP file transfers, emulating concurrent web browsing sessions or a BitTorrent download, for example.

This is a simplistic scenario for experimental purposes. Real VoIP applications, which we intend to evaluate in future work, often determine bit-rate based on network conditions. Real applications may also implement loss recovery mechanisms atop UDP, which may improve perceived voice quality when using UDP.

Latency: Figure 7 shows a CDF of one-way per-frame latency perceived by the receiving application, under heavy contention from 4 competing TCP streams. All three transports suffer major delays. 4% of UDP frames do not arrive at all, since UDP does not retransmit. 95% of frames sent with uCOBS over uTCP arrive within 200ms, compared to 80% of TCP frames.

Burst Losses: VoIP codecs such as SPEEX can interpolate across one or two missing frames, but are sensitive to burst losses or delays, which yield user-perceptible blackouts. An application's susceptibility to blackouts depends on its jitter buffer size: a larger buffer increases the receiver's tolerance of burst losses or delays, but also increases effective round-trip delay, which can add user-perceptible "lag" to all interactions.

The CDF in Figure 8 shows the prevalence of different lengths of burst losses experienced by the receiver in a typical VoIP call. A burst loss is a series of consecutive voice frames that miss their designated playout time, due either to loss or delay.

A 200ms jitter buffer of 3× the path RTT might seem generous, but the ITU's recommended maximum transmission time of 400ms [4] allows for a larger buffer with these network conditions. Now the differences between uCOBS and TCP are quite pronounced, with 80% of burst losses atop uCOBS being 3 or fewer packets, nearly matching that of UDP. Meanwhile 40% of TCP's bursts are greater than 10 packets, producing highly-perceptible 1/5-second pauses.

Perceptual Audio Quality: To illustrate the impact of unordered delivery on VoIP quality, we use Perceptual Evaluation of Speech Quality (PESQ) [32] to measure audio reproduction quality, by comparing the audio

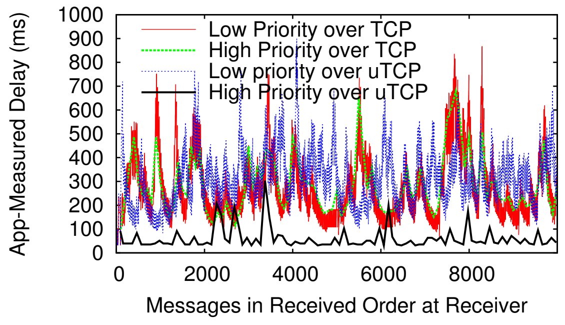


Figure 10: Prioritized messages experience lower end-to-end delay with uTCP.

stream reproduced by SPEEX at the receiver against that of an ideal run with no lost or delayed frames. We transmit a 4-minute VoIP call using a jitter buffer of 200ms, introducing 1 to 4 competing TCP streams progressively at 1-minute intervals.

Figure 9 plots PESQ quality scores for 2-second sliding time windows over a representative 4-minute call, comparing transmission via uCOBS, TCP, and UDP. The effect of network contention becomes apparent even with only one competing stream, but unordered delivery makes this impact much smaller on uCOBS or UDP than on TCP. uCOBS sometimes performs better than UDP, in fact, when uTCP successfully retransmits a lost segment within the jitter buffer's time window, whereas UDP never retransmits. (Some UDP applications employ application-level retransmission schemes [1], especially for control data.) Like TCP, uCOBS shows greater volatility than UDP with higher contention, due to TCP congestion control effects that uTCP preserves (though congestion control can be disabled). Similarly, the "back-off" of the *competing* streams enables the transports to rebound after the initial contention of 4 competing streams.

8.3 Send-Side Prioritization

To test uTCP's sender-side prioritization, we use a synthetic application that continuously sends messages to the receiver at network-limited rate, of which one in every 100 messages are considered "high-priority." Fig-

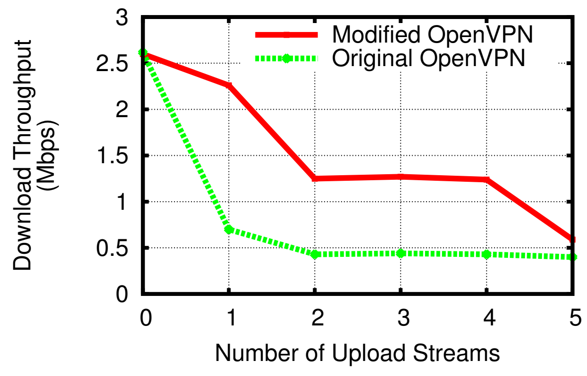


Figure 11: Throughput obtained by a TCP flow through modified and unmodified OpenVPN.

ure 10 plots application-observed messages delay over time, for high- and low-priority messages, atop TCP versus uTCP. As expected, high-priority messages consistently observe much lower delays under uTCP because they short-cut the TCP send queue. The next section explores a more realistic application for prioritization.

8.4 VPN Tunneling

Applications running atop TCP-based VPN tunnels often encounter *TCP-in-TCP* effects [48]. The applications' tunneled TCP flows assume they are running atop a best-effort, packet-switched network as usual, but are in fact running atop a reliable, in-order TCP-based tunnel. The TCP tunnel affects the tunneled flows' congestion control by increasing observed latency and RTT variance, and masks losses: tunneled flows never see "lost" or "reordered" TCP segments, only long-delayed ones. While uTCP does not change TCP's reliability or congestion control, it offers tunneled flows lower delay and jitter, and a more accurate view of packet losses.

To test Minion's impact on TCP-in-TCP effects, we made two changes to OpenVPN 2.1.4 [34]. First, we modified OpenVPN to use uCOBS instead of TCP, enabling unordered delivery of tunneled IP packets. Second, to reduce delay variance of tunneled TCP flows further, the modified OpenVPN gives tunneled TCP ACKs a higher priority than other packets.

The experiment uses a link with 3Mbps download and 0.5Mbps upload bandwidth, consistent with the median speed of residential connections [53].

Figure 11 shows measured throughput of a single *download*, with original and modified OpenVPN tunnels, for a varying number of competing *uploads* within the same tunnel. While using uTCP does not eliminate all TCP-in-TCP effects, the reduced RTT and RTT-variance noticeably improve performance.

To understand these performance improvements further, we now measure total network utilization achieved

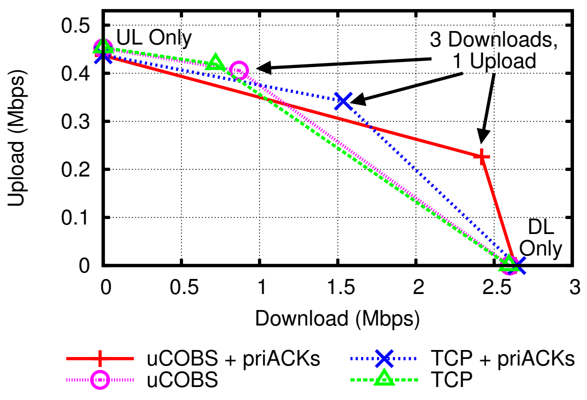


Figure 12: Contribution of independent modifications to network utilization.

by independently adding the two modifications—unordered delivery at the receiving ends of the tunnel (labeled *uCOBS*), and ACK prioritization at the sending ends (labeled *priACKs*)—leading to four variants of OpenVPN. Figure 12 shows total upload and download throughputs obtained by the VPN tunnel in three different contexts: one upload (labeled *UL*) within the tunnel, one upload competing with three downloads within the tunnel, and one download (labeled *DL*) in the tunnel.

With no competing flows, labeled *UL Only* and *DL Only* in Figure 12, all four variants perform similarly. With multiple competing downloads, out-of-order delivery improves download performance by a small amount, but ACK prioritization greatly improves download performance. Upload throughput suffers, however, as ACK prioritization is added. This throughput degradation is atrributable to the poor interaction between the small `write()`s of ACK packets being sent through the tunnel and Linux's `skbuff`-based congestion control described in Section 8.1. Despite this degradation to upload throughput, the area under the curve—representing total network utilization—remains highest with the fully modified tunnel. In a future version of uTCP that fixes this Linux-specific issue, we expect the upload throughput to remain high even with ack-prioritization, and network utilization to reflect uTCP's benefits more clearly.

8.5 Multistreaming Web Transfers

To explore uTCP's potential benefits for web browsing, we built msTCP, a simple *multistreaming* protocol providing multiple concurrent, ordered message streams atop a single uTCP connection. While similar in purpose to SPDY [2], to our knowledge msTCP is the first TCP-based multistreaming protocol that offers the unordered delivery benefits of non-TCP-based multistreaming protocols such as SCTP [45] and SST [19]. We omit a detailed description of msTCP for space reasons, but its design follows standard techniques.

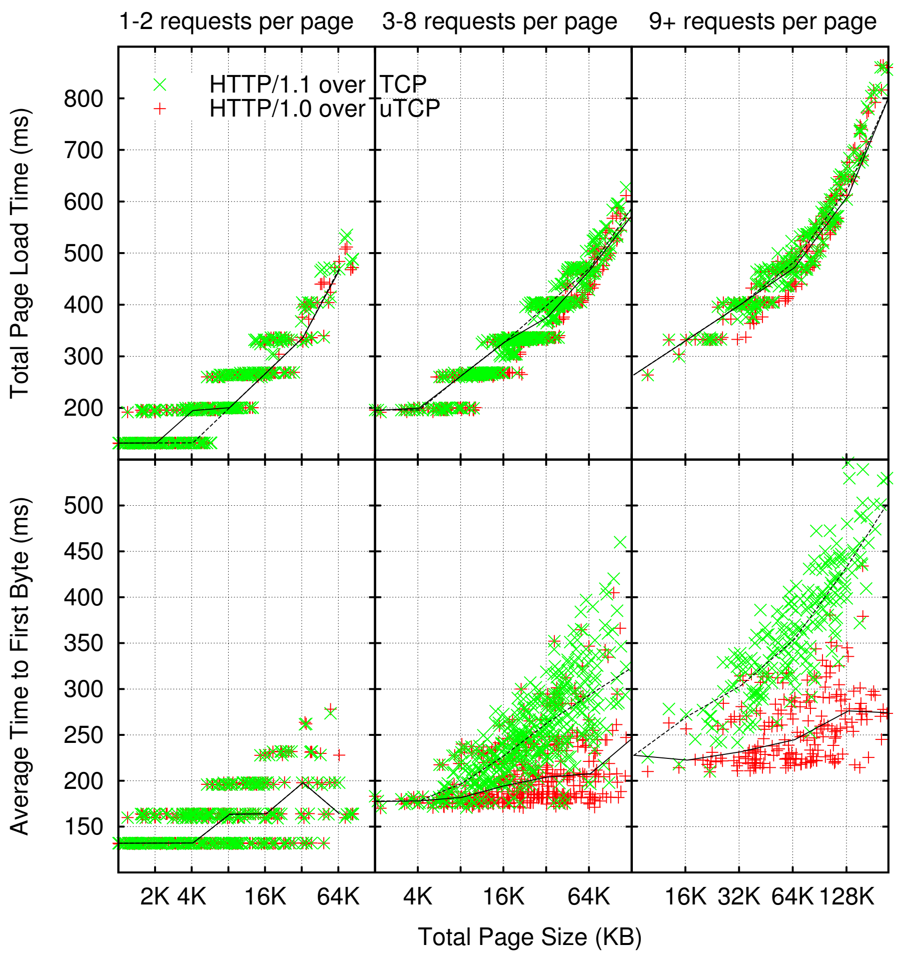


Figure 13: Pipelined HTTP/1.1 over a persistent TCP connection, vs. Parallel HTTP/1.0 over msTCP.

To evaluate msTCP, we compare the performance of parallel HTTP/1.0-style object requests over msTCP, against pipelined HTTP/1.1 requests on a persistent TCP connection, under a trace-driven web workload. We use a fragment of the UC Berkeley Home IP web client traces from the Internet Traffic Archive [25], using the trace to drive a series of web page downloads. Each page consists of a "primary" request for the HTML, followed by "secondary" requests for embedded objects such as images. The simulation pessimistically assumes that the browser cannot begin requesting secondary objects until it has downloaded the primary object completely, but at this point it can request all secondary objects in parallel. The experimental setup uses a link with 1.5Mbps bandwidth in each direction and with a 60ms RTT.

Figure 13 shows a scatter-plot of total page load time in the top three graphs, and in the bottom three graphs, average time to load the first byte of an object in each page—the time at which the browser can potentially start rendering the object. The dark curves show median times, computed across buckets of web page sizes. As the figure shows, msTCP does not affect total page load times noticeably. msTCP shows much lower delay in *starting* to load many objects, however, since msTCP interleaves different objects' chunks within the persistent connection.

Figure 13 shows the end-to-end impact on web browsing of msTCP's application-level message chunking and multiplexing in addition to the benefits of uTCP's out-of-order delivery. These latency savings, while not solely due to uTCP, represent the potential savings when web frameworks like SPDY [2] use uTCP, and make HTTP/HTTPS more usable as a general purpose substrate for deploying latency-sensitive applications [36].

8.6 Implementation Complexity

To evaluate the implementation complexity of uTCP and the related application-level code, Table 1 summarizes the source code changes uTCP makes to Linux's TCP stack in lines of code [15], the size of the standalone uCOBS library, and the changes uTLS makes to OpenSSL's `libssl` library. The SSL/TLS total does not

	TCP	uTCP	DCCP	SCTP
Kernel Code	12,982	**565** (4.6%)	6,338	19,312

	uCOBS	SSL/TLS	uTLS	DTLS
User Code	**732**	31,359	**586** (1.9%)	4,734

Table 1: Code size of uTCP prototype as a delta to Linux's TCP stack, the uCOBS library, and uTLS as a delta to `libssl` from OpenSSL. Code sizes of "native" out-of-order transports are included for comparison.

include OpenSSL's `libcrypt` library, which `libssl` requires but uTLS does not modify.

With only a 600-line change to the Linux kernel and less than 1400 lines of user-space support code, uTCP provides a delivery service comparable to Linux's 6,300-line native DCCP stack, while providing greater network compatibility. In user space, uTLS represents less than a 600-line change to the stream-oriented SSL/TLS protocol, contrasting with OpenSSL's 4,700-line implementation of DTLS, which runs only atop out-of-order transports such as UDP or DCCP.

9 Related Work

New transports for latency-sensitive apps: Brosh et al. [8] model TCP latency, and identify the regions of operation for latency-sensitive apps with TCP. While some of the considerations apply, such as latency induced by TCP congestion control, uTCP extends the working region for such apps by eliminating delays at the receiver.

DCCP [28, 29] provides an unreliable, unordered datagram service with negotiable congestion control. SCTP [45] provides unordered and partially-ordered delivery services to the application. Both DCCP and SCTP face large deployment barriers on today's Internet, however, and are thus not widely used.

New transports such as SST [19] and CUSP [47] run atop UDP to increase deployability, and UDP tunneling schemes have been proposed for standardized Internet transports as well [35, 49]. Many Internet paths block UDP traffic as well, however, as evidenced by the shift of popular VoIP applications such as Skype [7] and VPNs such as DirectAccess [16] toward tunneling atop TCP instead of UDP, despite the performance disadvantages.

Message Framing over TCP: Protocols such as HTTP [18], SIP [42], and iSCSI [43], can all benefit from out-of-order delivery, but use TCP for legacy and network compatibility reasons. All use simple type-length-value (TLV) encodings, which do not directly support out-of-order delivery even with uTCP, because they offer no reliable way to distinguish a record header from data in a TCP stream fragment. While COBS [12] represents an attractive set of characteristics for framing records to enable out-of-order delivery, other encodings such as BABS [10] also represent viable alternatives.

10 Conclusion

For better or worse, TCP remains the most common substrate for application-level protocols and frameworks, many of which can benefit from unordered delivery. Minion demonstrates that it is possible to obtain unordered delivery from wire-compatible TCP and TLS streams with surprisingly small changes to TCP stacks and application-level code. Without discounting the value of UDP and newer OS-level transports, Minion offers a more conservative path toward the performance benefits of unordered delivery, which we expect to be useful to applications that use TCP for a variety of pragmatic reasons.

Acknowledgments

We wish to thank Jeff Wise, Dishant Ailawadi, Stuart Cheshire, Matt Mathis, Kun Tan, and the anonymous reviewers for their valuable feedback and contributions to early drafts. This research was sponsored by the NSF under grants CNS-0916413 and CNS-0916678.

References

[1] OpenArena project. `http://openarena.ws/`.

[2] SPDY: An Experimental Protocol For a Faster Web. `http://www.chromium.org/spdy/spdy-whitepaper`.

[3] ZeroMQ: The intelligent transport layer. `http://www.zeromq.org`.

[4] ITU. Recommendation G.114: One-way transmission time, May 2003.

[5] F. Audet, ed. and C. Jennings. Network address translation (NAT) behavioral requirements for unicast UDP, Jan. 2007. RFC 4787.

[6] H. Balakrishnan, H. S. Rahul, and S. Seshan. An integrated congestion management architecture for Internet hosts. In *SIGCOMM*, Sept. 1999.

[7] S. A. Baset and H. Schulzrinne. An analysis of the Skype peer-to-peer Internet telephony protocol. In *INFOCOM*, Apr. 2006.

[8] E. Brosh, S. A. Baset, V. Misra, D. Rubenstein, and H. Schulzrinne. The delay-friendliness of TCP for real-time traffic. *IEEE Transactions on Networking*, 18(5):1478–1491, 2010.

[9] M. Carbone and L. Rizzo. Dummynet Revisited. *ACM CCR*, 40(2), Apr. 2010.

[10] J. S. Cardoso. Bandwidth-efficient byte stuffing. In *IEEE ICC 2007*, 2007.

[11] B. Carpenter and S. Brim. Middleboxes: Taxonomy and Issues, Feb. 2002. RFC 3234.

[12] S. Cheshire and M. Baker. Consistent Overhead Byte Stuffing. In *ACM SIGCOMM*, Sept. 1997.

[13] Cisco. Rate-Based Satellite Control Protocol, 2006.

[14] D. D. Clark and D. L. Tennenhouse. Architectural considerations for a new generation of protocols. In *SIGCOMM*, pages 200–208, 1990.

[15] A. Danial. Counting Lines of Code, ver. 1.53. `http://cloc.sourceforge.net/`.

[16] J. Davies. DirectAccess and the thin edge network. *Microsoft TechNet Magazine*, May 2009.

[17] T. Dierks and E. Rescorla. The transport layer security (TLS) protocol version 1.2, Aug. 2008. RFC 5246.

[18] R. Fielding et al. Hypertext transfer protocol — HTTP/1.1, June 1999. RFC 2616.

[19] B. Ford. Structured streams: a new transport abstraction. In *SIGCOMM*, Aug. 2007.

[20] B. Ford and J. Iyengar. Breaking up the transport logjam. In *HotNets-VII*, Oct. 2008.

[21] B. Ford and J. Iyengar. Efficient cross-layer negotiation. In *HotNets-VIII*, Oct. 2009.

[22] S. Guha, Ed., K. Biswas, B. Ford, S. Sivakumar, and P. Srisuresh. NAT behavioral requirements for TCP, Oct. 2008. RFC 5382.

[23] L. Guo, E. Tan, S. Chen, Z. Xiao, O. Spatscheck, and X. Zhang. Delving into Internet Streaming Media Delivery: a Quality and Resource Utilization Perspective. In *IMC*, Oct. 2006.

[24] M. Honda, Y. Nishida, C. Raiciu, A. Greenhalgh, M. Handley, and H. Tokuda. Is it still possible to extend TCP? In *IMC*, Nov. 2011.

[25] The Internet traffic archive. `http://ita.ee.lbl.gov/`.

[26] J. Iyengar, S. O. Amin, M. F. Nowlan, N. Tiwari, and B. Ford. Minion: Unordered delivery wire-compatible with TCP and TLS (full version), Apr. 2012. Technical Report. `arXiv:1103.0463`.

[27] S. Kent and K. Seo. Security architecture for the Internet protocol, Dec. 2005. RFC 4301.

[28] E. Kohler, M. Handley, and S. Floyd. Datagram congestion control protocol (DCCP), Mar. 2006. RFC 4340.

[29] E. Kohler, M. Handley, and S. Floyd. Designing DCCP: Congestion control without reliability. In *SIGCOMM*, 2006.

[30] J. Mogul. TCP offload is a dumb idea whose time has come. In *HotOS IX*, May 2003.

[31] P. Natarajan et al. SCTP: An innovative transport layer protocol for the Web. In *15th WWW*, May 2006.

[32] I.-T. T. S. S. of ITU. Wideband extension to recommendation p.862 for the assessment of wideband telephone networks and speech codecs, Nov. 2007.

[33] OpenSSL project. `http://www.openssl.org/`.

[34] OpenVPN project. `http://openvpn.net/`.

[35] T. Phelan. DCCP Encapsulation in UDP for NAT Traversal (DCCP-UDP), Aug. 2010. Internet-Draft draft-ietf-dccp-udpencap-02 (Work in Progress).

[36] L. Popa, A. Ghodsi, and I. Stoica. HTTP as the narrow waist of the future Internet. In *HotNets-IX*, Oct. 2010.

[37] J. Postel. User datagram protocol, Aug. 1980. RFC 768.

[38] C. Reis et al. Detecting in-flight page changes with web tripwires. In *5th NSDI*, Apr. 2008.

[39] E. Rescorla. HTTP over TLS, May 2000. RFC 2818.

[40] E. Rescorla and N. Modadugu. Datagram transport layer security, Apr. 2006. RFC 4347.

[41] J. Rosenberg. UDP and TCP as the new waist of the Internet hourglass, Feb. 2008. Internet-Draft (Work in Progress).

[42] J. Rosenberg et al. SIP: session initiation protocol, June 2002. RFC 3261.

[43] J. Satran, K. Meth, C. Sapuntzakis, M. Chadalapaka, and E. Zeidner. Internet small computer systems interface (iSCSI), Apr. 2004. RFC 3720.

[44] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. RTP: A transport protocol for real-time applications, July 2003. RFC 3550.

[45] R. Stewart, ed. Stream control transmission protocol, Sept. 2007. RFC 4960.

[46] Transmission control protocol, Sept. 1981. RFC 793.

[47] W. W. Terpstra, C. Leng, M. Lehn, and A. P. Buchmann. Channel-based unidirectional stream protocol (CUSP). In *INFOCOM Mini Conference*, Mar. 2010.

[48] O. Titz. Why TCP over TCP is a bad idea, Apr. 2001. `http://sites.inka.de/bigred/devel/tcp-tcp.html`.

[49] M. Tuexen and R. Stewart. UDP Encapsulation of SCTP Packets, Jan. 2010. Internet-Draft draft-tuexen-sctp-udp-encaps-05 (Work in Progress).

[50] J.-M. Valin. The speex codec manual version 1.2 beta 3, Dec. 2007. `http://www.speex.org/`.

[51] D. Velten, R. Hinden, and J. Sax. Reliable data protocol, July 1984. RFC 908.

[52] W3C. The websocket api (draft), 2011. `http://dev.w3.org/html5/websockets/`.

[53] www.speedmatters.org. 2010 report on internet speeds in all 50 states, Nov. 2010. `http://www.speedmatters.org/content/internet-speed-report`.

[54] M. Zec, M. Mikuc, and M. Zagar. Estimating the impact of interrupt coalescing delays on steady state TCP throughput. In *SoftCOM*, 2002.

How Hard Can It Be? Designing and Implementing a Deployable Multipath TCP

Costin Raiciu[†], Christoph Paasch[‡], Sebastien Barre[‡], Alan Ford,
Michio Honda[◇], Fabien Duchene[‡], Olivier Bonaventure[‡] and Mark Handley[⋆]

[†]Universitatea Politehnica Bucuresti, [‡]Universite Catholique de Louvain
[◇]Keio University, [⋆]University College London

ABSTRACT

Networks have become multipath: mobile devices have multiple radio interfaces, datacenters have redundant paths and multihoming is the norm for big server farms. Meanwhile, TCP is still only single-path.

Is it possible to extend TCP to enable it to support multiple paths for current applications on today's Internet? The answer is positive. We carefully review the constraints—partly due to various types of middleboxes—that influenced the design of Multipath TCP and show how we handled them to achieve its deployability goals.

We report our experience in implementing Multipath TCP in the Linux kernel and we evaluate its performance. Our measurements focus on the algorithms needed to efficiently use paths with different characteristics, notably send and receive buffer tuning and segment reordering. We also compare the performance of our implementation with regular TCP on web servers. Finally, we discuss the lessons learned from designing MPTCP.

1. INTRODUCTION

In today's Internet, servers are often multi-homed to more than one Internet provider, datacenters provide multiple parallel paths between compute nodes, and mobile hosts have multiple radios. Traditionally, it was the role of routing to take advantage of path diversity, but this has limits to responsiveness and scaling. To really gain both robustness and performance advantages, we need transport protocols engineered to utilize multiple paths. Multipath TCP[5] is an attempt to extend the TCP protocol to perform this role. At the time of writing, it is in in the final stage of standardization in the IETF.

Multipath TCP stripes data from a single TCP connection across multiple subflows, each of which may take a different path through the network. A linked congestion control mechanism[23] controls how much data is sent on each subflow, with the goal of explicitly moving traffic off the more congested paths onto the less congested ones. This paper is not about congestion control, but rather it is about the design of the Multipath TCP protocol itself. In principle, extending TCP to use multiple paths is not difficult, and there are a number of obvious ways in which it could be done. Indeed it was first proposed by Christian Huitema in 1995[11]. In practice though, the existence of middleboxes greatly constrains the design choices. The challenge is to make Multipath TCP not only robust to path failures, but also robust to failures in the presence of middleboxes that attempt to optimize single-path TCP flows. No previous extension to the core Internet protocols has needed to consider this issue to nearly the same extent.

In the first half of this paper we examine the design options for multipath TCP, with the aim of understanding both the end-to-end problem and the end-to-middle-to-end constraints. We use the results of a large Internet study to validate these design choices.

Designing MPTCP turned out to be more difficult than expected. For instance, a key question concerns how MPTCP metadata should be encoded — embed it in the TCP payload, or use the more traditional TCP options, with the potential for interesting interactions with middleboxes. In the IETF opinions were divided, with supporters on both sides. In the end, careful analysis revealed that MPTCP needs explicit connection level acknowledgments for flow control; further, these acknowledgments can create deadlocks if encoded in the payload. In reality, there was only one viable choice.

The second half of this paper concerns the host operating system. To be viable, Multipath TCP must be implementable in modern operating systems and must perform well. We examine the practical limitations the OS poses on MPTCP design and operation. This matters: our experiments show that one slow path can significantly degrade the throughput of the whole connection when MPTCP is underbuffered. We propose novel algorithms that increase throughput ten-fold in this case, ensuring MPTCP always matches what TCP would get on the best interface, regardless of buffer size.

It is not our goal to convince the reader that multipath transport protocols in general are a good idea. There has been a wealth of work that motivates the use of multipath transport for robustness[24], the use of linked congestion control across multiple paths for load balancing[23, 14, 4] and the ability of multi-path transport protocols to find and utilize unused network capacity in redundant topologies[10, 19]. Rather, the main contribution of this paper is the exploration of the design space for MPTCP confined by the many constraints imposed by TCP's original design, today's networks which embed TCP knowledge, and the need to perform well within the limitations imposed by the operating system.

2. GOALS

As many researchers have lamented, changing the behavior of the core Internet protocols is very difficult [7]. An idea may have great merit, but without a clear deployment path whereby the cost/benefit tradeoff for early adopters is positive, widespread adoption is unlikely.

We wish to move from a single-path Internet to one where the robustness, performance and load-balancing benefits of multipath transport are available to all applications, the majority of which use TCP for transport. To support such unmodified applications we must work below the sockets API, providing the same service as TCP: byte-oriented, reliable and in-order delivery. In theory we could use different protocols to implement this functionality as long as fallback to TCP is possible when one end does not support multipath. In practice there is no widely deployed signaling mechanism to select between transport protocols, so we have to use options in TCP's SYN exchange to negotiate new functionality.

The goal is for an unmodified application to start (what it believes to be) a TCP connection with the regular API. When both endpoints support MPTCP and multiple paths are available, MPTCP can set up additional subflows and stripe the connection's data across these subflows, sending most data on the least congested paths.

The potential benefits are clear, but there may be costs too. If negotiating MPTCP can cause connections to fail when regular TCP would have succeeded, then deployment is unlikely. The second goal, then, is for MPTCP to work in all scenarios where TCP currently works. If a subflow fails for any reason, the connection must be able to continue as long as another subflow has connectivity.

Third, MPTCP must be able to utilize the network at least as well as regular TCP, but without starving TCP. The congestion control scheme described in [23] meets this requirement, but congestion control is not the only factor that can limit throughput.

Finally MPTCP must be implementable in operating systems without using excessive memory or processing. As we will see, this requires careful consideration of both fast-path processing and overload scenarios.

3. DESIGN

The five main mechanisms in TCP are:

- Connection setup handshake and state machine.
- Reliable transmission & acknowledgment of data.
- Congestion control.
- Flow control.
- Connection teardown handshake and state machine.

The simplest possible way to implement Multipath TCP would be to take segments coming out of the regular stack and "stripe" them across the available paths *somehow*[1]. For this to work well, the sender would need to know which paths perform well and which don't: it would need to measure per path RTTs to quickly and accurately detect losses. To achieve these goals, the sender must remember which segments it sent on each path and use TCP Selective Acknowledgements to learn which segments arrive. Using this information, the sender could drive retransmissions independently on each path and maintain congestion control state.

This simple design has one fatal flaw: on each path, Multipath TCP would appear as a discontinuous TCP bytestream, which will upset many middleboxes (our study shows that a third of paths will break such connections). To achieve robust, high performance multipath operation, we need more substantial changes to TCP, touching all the components listed above. Congestion control has been described elsewhere[23] so we will not discuss it further in this paper.

In brief, here is how MPTCP works. MPTCP is negotiated via new TCP options in SYN packets, and the endpoints exchange connection identifiers; these are used later to add new paths—subflows—to an existing connection. Subflows resemble TCP flows on the wire, but they all share a single send and receive buffer at the endpoints. MPTCP uses per subflow sequence numbers to detect losses and drive retransmissions, and connection-level sequence numbers to allow reordering at the receiver. Connection-level acknowledgements are used to implement proper flow control. We discuss the rationale behind these design choices below.

3.1 Connection setup

The TCP three-way handshake serves to synchronize state between the client and server[2]. In particular, initial sequence numbers are exchanged and acknowledged, and TCP options carried in the SYN and SYN/ACK packets are used to negotiate optional functionality.

MPTCP must use this initial handshake to negotiate multipath capability. An MP_CAPABLE option is sent in the SYN and echoed in the SYN/ACK if the server

[1]such *striping* needs additional mechanisms because both destination-based forwarding and network ECMP try hard not to stripe packets belonging to the same TCP connection

[2]The correct terms should be *active opener* and *passive opener*. For conciseness, we use the terms client and server, but we do not imply any additional limitations on TCP usage.

understands MPTCP and wishes to enable it. Although this form of extension has been used many times, the Internet has grown a great number of middleboxes in recent years. Does such a handshake still work?

We performed a large study to test this - complete results are available in [9]. Our code generates specific TCP segments with the aim of testing what really happens on Internet paths. These tests were run from 142 access networks in 24 countries, including a wide mix of cellular providers, WiFi hotspots, home networks, as well as university and corporate networks. Although we cannot claim full coverage, the sample is large enough to provide good evidence for what does and what does not work in today's Internet.

We found that 6% of paths tested remove new options from SYN packets. This rises to 14% for connections to port 80 (HTTP). We did not observe any access networks that actually dropped a SYN with a new option. Most importantly, no path removed options from data packets unless it also removed them from the SYN, so it is possible to test a path using just the SYN exchange.

A separate study[3] probed Internet servers to see if new options in SYN packets caused any problems. Of the Alexa top 10,000 sites, 15 did not respond to a SYN packet containing a new option.

From these experiments we conclude that negotiating MPTCP in the initial handshake is feasible, but with some caveats. There is no real problem if a middlebox removes the MP_CAPABLE option from the SYN: MPTCP simply falls back to regular TCP behavior. However removing it from the SYN/ACK would cause the client to believe MPTCP is not enabled, whereas the server believes it is. This mismatch would be a problem if data packets were encoded differently with MPTCP. The obvious solution is to require the third packet of the handshake (ACK of SYN/ACK) to carry an option indicating that MPTCP was enabled. However this packet may be lost, so MPTCP must require all subsequent data packets to also carry the option until one of them has been acked. If the first non-SYN packet received by the server does not contain an MPTCP option, the server must assume the path is not MPTCP-capable, and drop back to regular TCP behavior.

Finally, if a SYN needs to be retransmitted, it would be a good idea to follow the retransmitted SYN with one that omits the MP_CAPABLE option.

It should be clear from this brief discussion of what should be the simplest part of MPTCP that anyone designing extensions to TCP must no longer think of the mechanisms as concerning only two parties. Rather, the negotiation is two-way *with mediation*, where the packets that arrive are not necessarily those that were sent. This requires a more defensive approach to protocol design than has traditionally been the case.

3.2 Adding subflows

Once two endpoints have negotiated MPTCP, they can open additional subflows. In an ideal world there would be no need to send new SYN packets before sending data on a new subflow - all that would be needed is a way to identify the connection that packets belong to. The strawman design simply sent TCP segments along different paths, and the endpoints used the 5-tuple to identify the proper connection. In practice though, we see that NATs and Firewalls rarely pass data packets that were not preceded by a SYN.

Adding a subflow raises two problems. First, the new subflow needs to be associated with an existing MPTCP flow. The classical five-tuple cannot be used as a connection identifier, as it does not survive NATs. Second, MPTCP must be robust to an attacker that attempts to add his own subflow to an existing MPTCP connection.

When the first MPTCP subflow is established, the client and the server insert 64-bit random keys in the MP_CAPABLE option. These will be used to verify the authenticity of new subflows.

To open a new subflow, MPTCP performs a new SYN exchange using the additional addresses or ports it wishes to use. Another TCP option, MP_JOIN is added to the SYN and SYN/ACKs. This option carries a MAC of the keys from the original subflow; this prevents blind spoofing of MP_JOIN packets from an adversary who wishes to hijack an existing connection. MP_JOIN also contains a connection identifier derived as a hash of the recipient's key [5]; this is used to match the new subflow to an existing connection.

If the client is multi-homed, then it can easily initiate new subflows from any additional IP addresses it owns. However, if only the server is multi-homed, the wide prevalence of NATs makes it unlikely that a new SYN it sends will be received by a client. The solution is for the MPTCP server to inform the client that the server has an additional address by sending an ADD_ADDR option on a segment on one of the existing subflows.

The client may then initiate a new subflow. This asymmetry is not inherent - there is no protocol design limitation that means the client cannot send ADD_ADDR or the server cannot send a SYN for a new subflow. But the Internet itself is so frequently asymmetric that we need two distinct ways, one implicit and one explicit, to indicate the existence of additional addresses.

3.3 Reliable multipath delivery

In a world without middleboxes, MPTCP could simply stripe data across the multiple subflows, with the sequence numbers in the TCP headers indicating the sequence number of the data in the connection in the normal TCP way. Our measurements show that this is infeasible in today's Internet:

- We observed that 10% of access networks rewrite TCP initial sequence numbers (18% on port 80). Some of this re-writing is by proxies that remove new options; a new subflow will fail on these paths. But many that rewrite do pass new options - these appear to be firewalls that attempt to increase TCP initial sequence number randomization. As a result, MPTCP cannot assume the sequence number space on a new subflow is the same as that on the original subflow.
- Striping sequence numbers across two paths leaves gaps in the sequence space seen on any single path. We found that 5% of paths (11% on port 80) do not pass on data after a hole - most of these seem to be proxies that block new options on SYNs and so don't present a problem as MPTCP is never enabled on these paths. But a few do not appear to be proxies, and so would stall MPTCP. Perhaps worse, 26% of paths (33% on port 80) do not correctly pass on an ACK for data the middlebox has not observed - either the ACK is dropped or it is "corrected".

Given the nature of today's Internet, it appears extremely unwise to stripe a single TCP sequence space across more than one path. The only viable solution is to use a separate contiguous sequence space for each MPTCP subflow. For this to work, we must also send information mapping bytes from each subflow into the overall data sequence space, as sent by the application. We shall return to the question of how to encode such mappings after first discussing flow control and acknowledgments, as the three are intimately related.

3.3.1 Flow control

TCP's receive window indicates the number of bytes beyond the sequence number from the acknowledgment field that the receiver can buffer. The sender is not permitted to send more than this amount of additional data.

Multipath TCP also needs to implement flow control, although packets now arrive over multiple subflows. If we inherit TCP's interpretation of receive window, this would imply an MPTCP receiver maintains a pool of buffering per subflow, with receive window indicating per-subflow buffer occupancy. Unfortunately such an interpretation can lead to a deadlock scenario:

1. The next packet that needs to be passed to the application was sent on subflow 1, but was lost.
2. In the meantime subflow 2 continues delivering data, and fills its receive window.
3. Subflow 1 fails silently.
4. The missing data needs to be re-sent on subflow 2, but there is no space left in the receive window, resulting in a deadlock.

The receiver could solve this problem by re-allocating subflow 1's unused buffer to subflow 2, but it can only do this by rescinding the advertised window on subflow 1. Besides, the receiver does not know which subflow the next packet will be sent on. The situation is made even worse because a TCP proxy[3] on the path may hold data for subflow 2, so even if the receiver opens its window, there is no guarantee that the first data to arrive is the retransmitted missing packet.

The correct solution is to generalize TCP's receive window semantics to MPTCP. For each connection a single receive buffer pool should be shared between all subflows. The receive window then indicates the maximum *data* sequence number that can be sent rather than the maximum subflow sequence number. As a packet resent on a different subflow always occupies the same data sequence space, no such deadlock can occur.

The problem for an MPTCP sender is that to calculate the highest data sequence number that can be sent, the receive window needs to be added to the highest data sequence number acknowledged. However the ACK field in the TCP header of an MPTCP subflow must, by necessity, indicate only *subflow* sequence numbers. Does MPTCP need to add an extra *data acknowledgment* field for the receive window to be interpreted correctly?

3.3.2 Acknowledgments

To correctly deduce a cumulative data acknowledgment from the subflow ACK fields, an MPTCP sender might keep a scoreboard of which data sequence numbers were sent on each subflow. However, the inferred value of the cumulative data ACK does not step in precisely the same way that an explicit cumulative data ACK would. Consider the sequence shown in Fig.1(a)[4]:

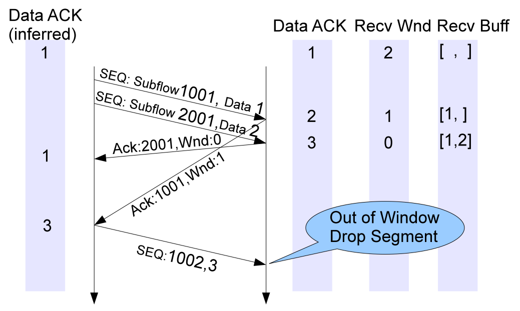
1. Data sequence no. 1 is sent on subflow 1 with subflow sequence number 1001.
2. Receiver sends ACK for 1001 on subflow 1.
3. Data sequence no. 2 is sent on subflow 2 with subflow sequence number 2001.
4. Receiver sends ACK for 2001 on subflow 2.
5. ACK for 2001 arrives (the RTT on subflow 2 is shorter).
6. ACK for 1001 arrives at sender.

The receiver expected the ACK for 1001 to be an implicit data ACK for 1, and the ACK for 2001 to be an implicit ACK for 2. However, as the ACK for 2001 does not implicitly acknowledge both 1 and 2, the sender's inferred data ACK is still 0 after step 5. Only after step 6 does the inferred data ACK become 2.

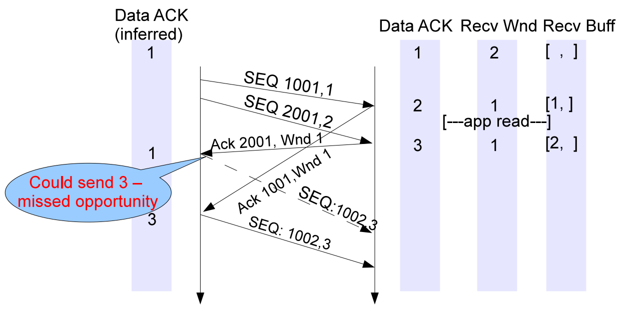
This sort of reordering is inevitable with multipath given the different RTTs of the different paths, and it would not by itself be a problem, except that the receiver needs to code the receive window field relative to the implicit data ACK. Suppose the receive buffer were only

[3]Most will prevent MPTCP being negotiated, but a few do not.

[4]The example uses packet sequence numbers for clarity, but MPTCP actually uses byte sequence numbers just like TCP



(a) Drops due to incorrect inference



(b) Stalls due to incorrect inference

Figure 1: Problems with inferring the cumulative data ACK from subflow ACK

two packets, and the application is slow to read. In the ACK for 1001, the receiver closes the receive window to one packet. In the ACK for 2001 the receiver closes the receive window completely, as there is no space remaining. When the ACK for 1001 is finally received, the inferred cumulative data ACK is now 2; the sender adds the receive window of one to this, and concludes incorrectly that the receiver has sufficient buffer space for one more packet. Fig. 1(b) shows a similar situation where reordering causes sending opportunities to be missed.

To avoid such scenarios MPTCP must carry an explicit data acknowledgment field, which gives the left edge of the receive window.

3.3.3 Encoding

We have seen that in the forward path we need to encode a mapping of subflow bytes into the data sequence space, and in the reverse path we need to encode cumulative data acknowledgments. There are two viable choices for encoding this additional data:

- Send the additional data in TCP options.
- Carry the additional data within the TCP payload, using a chunked or escaped encoding to separate control data from payload data.

For the forward path we have not found any compelling arguments either way, but the reverse path is a different matter. Consider a hypothetical encoding that divides the payload into chunks where each chunk has a TLV header. A data acknowledgment can then be embedded into the payload using its own chunk type. Under most circumstances this works fine. However, unlike TCP's pure ACK, anything embedded in the payload must be treated as data. In particular:

- It must be subject to flow control because the receiver must buffer data to decode the TLV encoding.
- If lost, it must be retransmitted consistently, so that middleboxes can track sequence state correctly[5]
- If packets before it are lost, it might be necessary to wait for retransmissions before the data can be parsed - causing head-of-line blocking.

[5]In our observations, the usual TCP proxies re-asserted the original content when sent a "retransmission" with different data. We also found one path that did this without exhibiting any other proxy behavior - this is symptomatic of a traffic normalizer[8] - and one on port 80 that reset the connection.

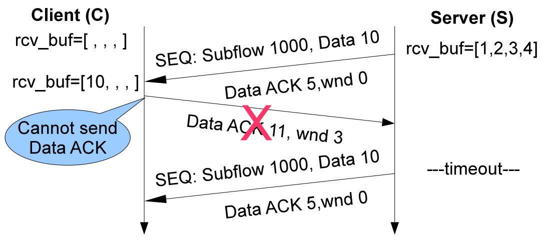


Figure 2: Flow Control on the path from C to S inadvertently stops the data flow from S to C

Flow control presents the most obvious problem for the chunked payload encoding. Figure 2 provides an example. Client C is pipelining requests to server S; meanwhile S's app is busy sending the large response to the first request so it isn't yet ready to read the subsequent requests. At this point, S's receive buffer fills up.

S sends segment 10, C receives it and wants to send the DATA ACK, but cannot: flow control imposed by S's receive window stops him. Because no DATA ACKs are received from C, S cannot free his send buffer, so this fills up and blocks the sending application on S. S's application will only read when it has finished sending data to C, but it cannot do so because its send buffer is full. The send buffer can only empty when S receives the DATA ACK from C, but C cannot send this until S's application reads. This is a classic deadlock cycle.

As no DATA ACK is received, S will eventually time out the data it sent to C and will retransmit it; after many retransmits the whole connection will time out.

It has been suggested that this can be avoided if DATA ACKs are simply excluded from flow control. Unfortunately any middlebox that buffers data can foil this; it is unaware the DATA ACK is special because it looks just like any other TCP payload.

When the return path is lossy, decoding DATA ACKs will be delayed until retransmissions arrive - this will effectively trigger flow control on the forward path, reducing performance. In effect, this would break MPTCP's goal of doing "no worse" than TCP on the best path.

Our conclusion is that DATA ACKs cannot be safely encoded in the payload. The only real alternative is to encode them in TCP options which (on a pure ACK packet) are not subject to flow control.

3.3.4 Data sequence mappings

If MPTCP must use options to encode DATA ACKs, it is simplest to also encode the mapping from subflow sequence numbers to data sequence numbers in a TCP option. This is the *data sequence mapping* or DSM.

At first we thought that the DSM option simply needed to carry the data sequence number corresponding to the start of the MPTCP segment. Unfortunately middleboxes and "smart" NICs make this far from simple.

Middleboxes that resegment data would cause a problem. [6] TCP Segmentation Offload (TSO) hardware in the NIC also resegments data and is commonly used to improve performance. The basic idea is that the OS sends large segments and the NIC resegments them to match the receiver's MSS. What does TSO do with TCP options? We tested 12 NICs supporting TSO from four different vendors. All of them copy a TCP option sent by the OS on a large segment into all the split segments.

If MPTCP's DSM option only listed the data sequence number, TSO would copy the same DSM to more than one segment, breaking the mapping. Instead the DSM option must say precisely which subflow bytes map to which data sequence numbers. But this is further complicated by middleboxes that rewrite sequence numbers; these are commonplace — 10% of paths. Instead, the DSM option must map the offset from the subflow's initial sequence number to the data sequence number, as the offset is unaffected by sequence number rewriting. The option must also contain the length of the mapping. This is robust - as long as the option is received, it does not greatly matter which packet carries it, so duplicate mappings caused by TSO are not a problem.

3.3.5 Send buffer management

The sender will free segments from the connection-level send queue only when they are acknowledged by a DATA ACK. Even if a segment is ACKed at the subflow level, its data is kept in memory until we receive a DATA ACK. If a DATA ACK does not arrive, a timer fires and the sender retransmits that data. This allows the receiver to ACK all segments correctly received at the subflow level, which in turn allows the sender to correctly infer path properties. This separation of functionality also allows the receiver to drop data that is in-window at the subflow level but out-of-window at the connection level.

[6]We did not observe any that would both permit MPTCP and resegment, though.

Further, if a middlebox coalesces packets, TCP's limited option space means it can only keep one of the data sequence mapping options on the coalesced segment. The receiver will get a bigger segment where some of the bytes have no mapping. The packet will be acknowledged at the subflow-level, and only the bytes with the mapping will be acknowledged at the data level. This causes the sender to retransmit the missing bytes, allowing the MPTCP connection to make progress.

3.3.6 Content-modifying middleboxes

Many NAT devices include application-level gateway functionality for protocols such as FTP: IP addresses and ports in the FTP control channel are re-written to correct for the address changes imposed by the NAT.

Multipath TCP and such content-modifying middleboxes have the potential to interact badly. In particular, due to FTP's ASCII encoding, re-writing an IP address in the payload can necessitate changing the length of the payload. Subsequent sequence and ACK numbers are then fixed up by the middlebox so they are consistent from the point of view of the end systems.

Such length changes break the DSM option mapping - subflow bytes can be mapped to the wrong place in the data stream. They also break every other mapping mechanism we considered, including chunked payloads. There is no easy way to handle such middleboxes.

After much debate, we concluded that MPTCP must include a checksum in the DSM mapping so such content changes can be detected. MPTCP rejects a modified segment and triggers a fallback process: if any other subflows exists, MPTCP terminates the subflow on which the modification occurred; if no other subflow exists, MPTCP drops back to regular TCP behavior for the remainder of the connection, allowing the middlebox to perform rewriting as it wishes.

Calculating a checksum over the data is comparatively expensive, and we did not wish to slow down MPTCP just to catch such rare corner cases. MPTCP therefore uses the same 16-bit ones complement checksum used in the TCP header. This allows the checksum over the payload to be calculated only once. The payload checksum is added to a checksum of an MPTCP pseudo header covering the DSM mapping values and then inserted into the DSM option. The same payload checksum is added to the checksum of the TCP pseudo-header and then used in the TCP checksum field.

With this mechanism a software implementation incurs little additional cost from calculating the MPTCP checksum. Unfortunately, modern NICs frequently perform checksum offload. If the TCP stack uses the NIC to calculate checksums, with MPTCP it will still need to calculate the MPTCP checksum in software, negating the benefits of checksum offload. There is little we can do about this, other than to note that future NICs will likely perform MPTCP checksum offload too, if MPTCP is widely deployed. In the meantime, MPTCP allows checksums to be disabled for high performance environments such as data-centers where there is no chance of encountering such an application-level gateway.

The fallback-to-TCP process, triggered by a checksum failure, can also be triggered in other circumstances. For example, if a routing change moves an MPTCP subflow to a path where a middlebox removes DSM options, this also triggers the fallback procedure.

3.4 Connection and subflow teardown

TCP has two ways to indicate connection shutdown: FIN for normal shutdown and RST for errors such as when one end no longer has state. With MPTCP, we need to distinguish subflow teardown from connection teardown. With RST, the choice is clear: it must only terminate the subflow, or an error on a single subflow would cause the whole connection to fail.

Normal shutdown is slightly more subtle. TCP FINs occupy sequence space; the FIN/FIN-ACK/ACK handshake and the cumulative nature of TCP's acknowledgments ensure that not only all data has been received, but also both endpoints know the connection is closed and know who needs to hold TIMEWAIT state.

How then should a FIN on an MPTCP subflow be interpreted? Does it mean that the sending host has no more data to send, or only that no more data will be sent on this subflow? Another way to phrase this is to ask whether a FIN on a subflow occupies data sequence space, or just subflow sequence space?

Consider first what would happen if a FIN occupied data sequence space. This could be achieved by extending the length of the DSM mapping in a packet to cover the FIN. Mapping the FIN into the data sequence space in this way tells the receiver what the data sequence number of the last byte of the connection is, and hence whether any more data is expected from other subflows.

Suppose that some data had been transmitted on subflow A just before the last data and FIN were sent on subflow B. If the receiver is really unlucky, subflow A may fail (perhaps due to mobility) before the last data arrives. When the sender times out this data, it will wish to re-send it on subflow B, but it has already sent a FIN on this subflow. Sending data after the FIN is sure to confuse middleboxes and firewalls that tore down state when they observed the FIN. This problem might be avoided by delaying sending the FIN until all outstanding data has been DATA ACKed, but this adds an unnecessary RTT to all connections during which the receiving application doesn't know if more data will arrive.

Much simpler is for a FIN to have the more limited "no more data on this subflow" semantics, and this is what MPTCP does. An explicit DATA FIN, carried in a TCP option, indicates the end of the data sequence space and can be sent immediately when the application closes the socket. To be safe, either the sender waits for the DATA ACK of the DATA FIN before sending a FIN on each subflow, or it sends DATA FIN on all subflows together with a FIN.

MPTCP's FIN semantics also allow subflows to be closed cleanly while allowing the connection to continue on other subflows. Finally, to support mobility, MPTCP provides a REMOVE_ADDR message, allowing one subflow to indicate that other subflows using the specified address are closed. This is necessary to cleanly cope with mobility when a host loses the ability to send from an address and so cannot send a subflow FIN.

4. IMPLEMENTATION ISSUES

To validate the design of MPTCP and understand its impact on real applications, we added full support for MPTCP to version 2.6.38 of the Linux kernel. This is a major modification to TCP: our patch to the Linux kernel, available from http://mptcp.info.ucl.ac.be, is about 10,400 lines of code . The software architecture is described in detail in [1]. To our knowledge, this is the first full kernel implementation of MPTCP.

We will focus on three of the more recent important improvements to the MPTCP implementation. We first briefly describe the algorithms that have been included in our MPTCP implementation to deal with middleboxes. Then we explain how to reduce the MPTCP memory usage. Finally we show how an MPTCP receiver is able to handle out-of-order data efficiently.

4.1 Supporting middleboxes

As we have seen, middleboxes constrain the design of MPTCP in many ways. To verify whether our design and its implementation are robust to middleboxes, we implemented Click elements [15] that model the various operations performed by middleboxes, namely:

- NAT
- Sequence number rewriting
- Removing TCP options
- Payload modification
- Segment splitting
- Segment coalescing
- Pro-active acking

The simplest middleboxes are NATs and those that rewrite sequence numbers: beyond implementing the basic MPTCP design, no special code is required to support them. Middleboxes may also remove TCP options. If a middlebox removes the MP_CAPABLE option from

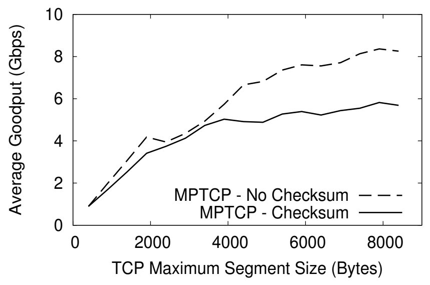


Figure 3: Impact of enabling or disabling DSM checksums in 10G environments.

the SYN or SYN/ACK, MPTCP is not used for the connection. If a middlebox removes the MPTCP option from non-SYN segments, our implementation falls back to regular TCP and continues the data transfer.

We also considered the impact of middleboxes that split or coalesce segments. NICs that support TCP Segmentation Offload (TSO) are an example of the former and traffic normalizers [8] are an example of the latter. Our implementation supports both. However, coalescing middleboxes cause a performance degration due to the loss of data sequence mappings that force the sender to retransmit data. In reality though, we have not observed any middleboxes that coalesce segments with unknown options.

Application-level gateways[22] are the most difficult middleboxes to support; they modify the payload and adjust TCP sequence numbers to compensate. MPTCP uses the DSM checksum to detect these. If we detect a DSM-checksum failure on only one subflow, that subflow is reset and the transfer continues on another subflow. If the middlebox affects all subflows, our implementation falls back to regular TCP.

Unfortunately, calculating checksums may affect performance. To evaluate this impact, we used Xeon class servers attached to 10 Gbps Ethernet interfaces. Figure 3 shows the MPTCP goodput as a function of MSS. Checksum offloading is not yet supported in our code, so the per-packet checksums are computed in software. With the default Ethernet MSS, the performance is limited by per-packet costs such as interrupt processing.

As the MSS increases, the fixed per-packet costs have less impact and goodput increases. When DSM checksums are switched off, our implementation uses the NIC to offload checksum calculations at the sender and receiver. When DSM checksums are enabled, the sender must calculate the DSM checksum in software and the receiver must check it. With jumbo frames, these checksums reduce throughput by 30%.

4.2 Minimizing memory usage

TCP and MPTCP provide in-order, reliable delivery of data to the application. The network can reorder packets or lose them, so the receiver must buffer out-of-order packets before sending a cumulative ACK and passing them to the application. Consequently, the sender also allocates a similar sized pool of memory to hold in flight segments until they are acknowledged.

How big must the receive buffer be for TCP to work well? In the absence of loss, a bandwidth-delay product (BDP) of buffering is needed to avoid flow control. If, after a packet loss, we want the sender to be able to keep sending packets while in fast retransmit we need an extra BDP of receive buffer.

For MPTCP the story is a bit different. Assuming there are no losses, and no special scheduling at the sender, the receive buffer must be at least $\sum x_i RTT_{max}$ where x_i is the throughput of subflow i and RTT_{max} is the highest RTT of all the subflows. This allows all paths to keep sending while waiting for an early packet to be delivered on the slowest path. If we want to allow all paths to keep sending while any path is fast retransmitting, the buffer must be doubled: $2\sum x_i RTT_{max}$.

We first observe that, fundamentally, memory requirements for MPTCP are much higher than those for TCP, mostly because of the RTT_{max} term. A 3G path with a bandwidth of 2 Mbps and 150 ms RTT needs just 75 KB of receive-buffer, while a WiFi path running at 8 Mbps with 20 ms RTT needs around 40 KB. MPTCP running on the same two paths will need 375 KB — nearly four times the sum of the path BDPs.

We used our Linux implementation to test this issue. Fig. 4(a) shows the throughput achieved as a function of receive-window for TCP and MPTCP running over an emulated 8Mbps WiFi-like path (base RTT 20ms, 80ms buffer) and an emulated 2Mbps 3G path (base RTT 150ms, 2s buffer).

MPTCP will send a new packet on the lowest delay link that has space in its congestion window. When there is very little receive buffer, MPTCP sends all packets over WiFi, matching regular TCP. With a bigger buffer, additional packets are put on 3G and overall throughput drops. Somewhat surprisingly, even 370KB are insufficient to fill both pipes. This is because unnecessarily many packets are sent over 3G. This pushes the effective RTT_{max} towards 2 seconds, and so the receive buffer needed to avoid flow control increases.

We see that a megabyte of receive-buffer (and send-buffer) are needed for a single connection over 3G and WiFi. This is a problem, and may prevent MPTCP from being used on busy servers and memory-scarce mobile phones. TCP over WiFi even outperforms MPTCP over both WiFi and 3G when the receive buffer is less than 400KB, removing any incentive to deploy MPTCP.

In the rest of this section we outline a series of mechanisms to be implemented at the sender that allow MPTCP make the most of the memory it has available. Solutions

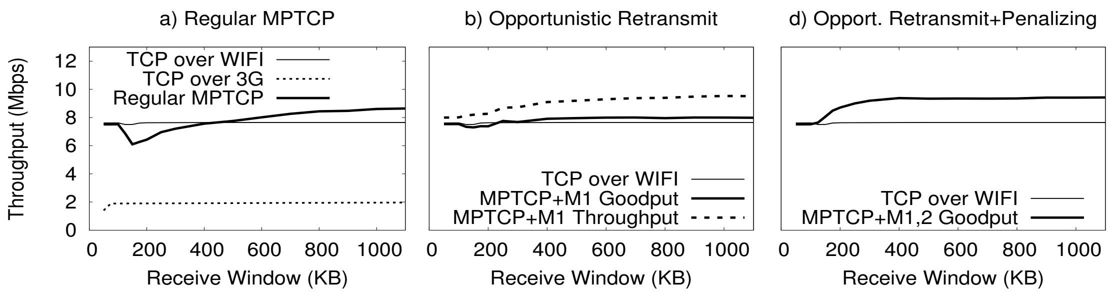


Figure 4: Receive buffer impact on throughput

have to be adaptive: as more receive buffer becomes available, MPTCP should use up more of the capacity it has available. This way, if the OS is prepared to spend the memory it will achieve higher throughput; if not, it will receive the same as TCP.

Mechanism 1: Opportunistic retransmission. When a subflow has sufficient congestion window to send more packets, but there is no more space in the receive window, what should it do? One option is to resend the data, previously sent on another subflow, that is holding up the trailing edge of the receive window. In our example, the WiFi subflow may retransmit some data unacknowledged data sent on the slow 3G subflow.

The motivation is that this allows the fast path to send as fast as it would with single-path TCP, even when underbuffered. If the connection is not receive-window limited, opportunistic retransmission never gets triggered.

Our Linux implementation only considers the first unacknowledged segment to avoid the performance penalty of iterating the potentially long send-queue in software interrupt context. This works quite well by itself, as shown in Fig. 4(b): MPTCP throughput is almost always as good as TCP over WiFi, and mostly it is better.

Unfortunately opportunistic retransmission is rather wasteful of capacity when underbuffered, as it unnecessarily pushes 2Mbps traffic over 3G; this accounts for the difference between goodput and throughput in Fig.4(b).

Mechanism 2: Penalizing slow subflows. Reacting to receive window stalls by retransmitting is costly; we'd prefer a way to avoid persistently doing so. If a connection has just filled the receive window, to avoid doing so again next RTT we need to reduce the RTT on the subflow that is holding up the advancement of the window. To do this, MPTCP can reduce that subflow's window; in our tests we halved the congestion window and set the slowstart threshold to the reduced window size. To avoid repeatedly penalizing the same flow, only one reduction is applied per subflow round-trip time.

Penalizing and opportunistic retransmission work well together, as seen in Fig. 4(c): MPTCP always outperforms or a least matches TCP over WiFi.

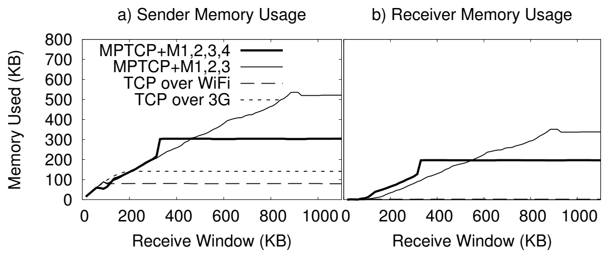


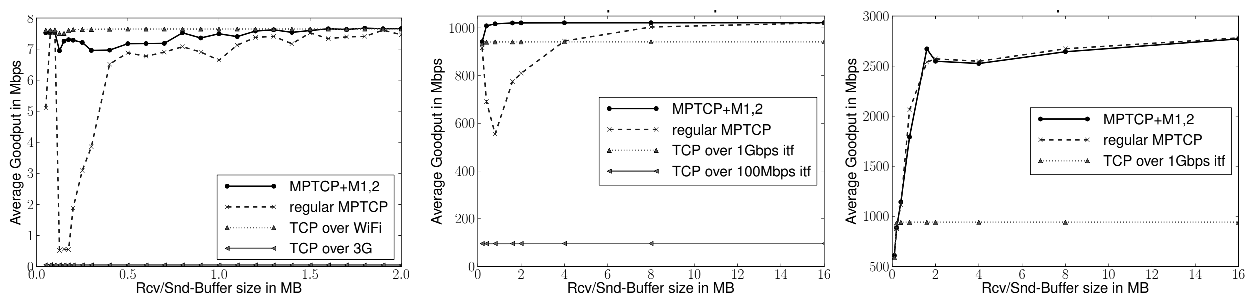
Figure 5: Receive buffer impact on memory use

Mechanisms 3 & 4: Buffer autotuning with capping. Taken together, mechanisms 1 & 2 allow an MPTCP sender to effectively utilize whatever receive buffer the receiver makes available. However, modern TCP implementations don't just blindly allocate large buffers from the outset - they adaptively increase the buffer size as they discover more buffer is needed. We've implemented both send and receive buffer tuning, done using the MPTCP buffer size formula above. In our experiments, we set the maximum send and receive buffers with the usual sysctls, but it is autotuning that automatically increases the actual buffer.

With TCP it is generally safe to configure large maximum send and receive buffer sizes; autotuning ensures they won't be used unless they are really needed. With MPTCP, however, if one of the subflows is on a path with excessive network buffering, as is common with 3G providers, autotuning will measure a large value for RTT_{max} and ramp up the receive buffer size unnecessarily. Mechanisms 1 & 2 only kick in once the receive buffer has grown and then been filled.

To see the effect of buffer autotuning, in Fig. 5 we use *htsim* to simulate the average memory consumption as a function of configured maximum receive buffer for WiFi and 3G. Memory consumption at the sender is lowest for TCP over WiFi, where the BDP is lowest. TCP over 3G has higher consumption, and MPTCP uses up to 500KB when the configured receive-buffer permits it.

This is more than MPTCP really needs; most of the time it is unnecessary to fill the large buffers on the 3G link. To avoid this effect, we might *cap* the congestion window when the amount of data buffered is above one



(a) WiFi and a lossy 3G path (b) 1Gbps and 100Mbps links (c) Three 1Gbps links

Figure 6: The Receive-buffer optimizations significantly improve goodput with small buffers

BDP. This is easy to implement: measure the base RTT of the subflow's path by taking the min of all RTT measurements, and cap $cwnd$ when the smoothed RTT is double the base RTT. In this simulation, capping halves MPTCP's memory usage when the configured receive buffer is large. Our Linux implementation does not yet support capping, but FreeBSD's regular TCP implementation has supported this since 2002, enabled via the net.inet.tcp.inflight.enable sysctl[6].

Despite the large advertised receive window, actual memory consumption at the receiver is small for single path TCP on both 3G and WiFi so long as losses are rare and the receiving application reads as soon as data is available. The same is not true for MPTCP: the receiver will spend at least two thirds of the memory the sender spends, due to reordering induced by the use of multiple paths. The effect is pronounced in the example we chose, where the difference between WiFi and 3G RTTs is seven-fold. For equal delay paths, MPTCP's receiver memory consumption is also close to zero.

4.2.1 Further evaluation

To evaluate our algorithms, we used both our *htsim* simulator and our Linux kernel implementation. We used simulation to test the viability of our proposals and their sensitivity to a wide range of path properties. The sensitivity analysis showed that the algorithms are robust and work well in a wide range of scenarios. We also tested MPTCP competing with single path TCP flows and found that MPTCP does get the same throughput as TCP on the best path or strictly better in the vast majority of cases. MPTCP does underperform TCP by 20%-30% when the best subflow experiences frequent timeouts; however, this is not caused by the receive-buffer algorithms, but by MPTCP's congestion controller overestimating the throughput of subflows that experience loss rates of greater than 10%.

To illustrate more clearly the impact of mechanisms 1 and 2, it is worth examining a few more varied scenarios.

The first scenario we analyze is where one of the paths has extremely poor performance such as when mobile devices have very weak signal. Figure 6(a) shows throughput achieved using our Linux implementation on an emulated WiFi path (8Mbps, 20ms RTT, 80ms buffer) and an emulated very slow 3G link (50Kbps, 150ms RTT, 2s buffer). As the link is so slow, the loss rate will be high on the 3G path, and the large network buffer means that retransmission over 3G takes a long time. With receive buffer sizes of less than 400KB, whenever a loss happens on 3G, regular MPTCP ends up flow-controlled, unable to send on the fast WiFi path. MPTCP plus mechanisms 1 and 2 is able to avoid this being a persistent problem. Opportunistic retransmission allows the lost 3G packet to be re-sent on WiFi without waiting for a timeout and penalization reduces the data buffered on the 3G link, avoiding the situation repeating too quickly. With receive buffer sizes around 200KB, these mechanisms increase MPTCP throughput tenfold.

Next, we use two hosts connected by one gigabit and one 100Mb/s link to emulate inter-datacenter transfers with asymmetric links. Fig. 6(b) shows that MPTCP+M1,2 is able to utilize both links using only 250KB of memory, while regular MPTCP underperforms TCP over the 1Gbps interface until the receive buffer is at least 2MB.

When the hosts are connected via symmetric links—we used three such links in Figure 6(c)—both regular MPTCP and MPTCP+M1,2 perform equally well, regardless of the receive buffer size. This is because in this scenario, when underbuffered, using the fastest path is the optimal strategy.

Application level latency Goodput is not the only metric that is important for applications. For interactive applications, latency between the sending application and the receiving application can matter.

As MPTCP uses several subflows with different RTTs, we expect it to increase the end-to-end latency seen by the application compared to TCP on the fastest path. To test this, we use an application that sends 8 KByte blocks of data and timestamps each block's transmission and reception. This allows us to measure the variation of the end-to-end delay as seen by the application.

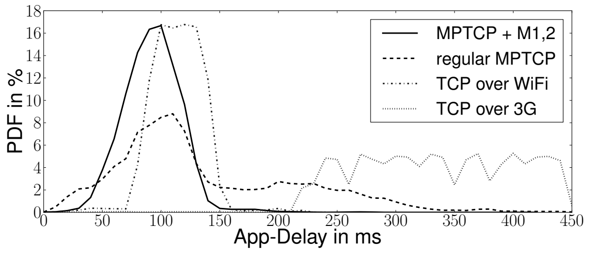


Figure 7: Application level latency for 3G/WiFi case

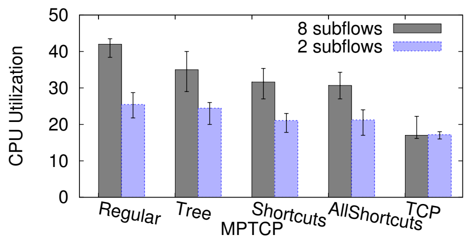


Figure 8: Effect of ofo receive algorithms on load

Figure 7 shows the probability density function of the application-delay with a buffer-size of 200KB running over 3G and WiFi. Mechanisms 1 and 2 do a good job of avoiding the larger latencies seen with regular MPTCP. Somewhat counter intuitively, the latency of TCP over WiFi is actually greater than MPTCP+M1,2. The reason for this is that 200KB is more buffering than TCP needs over this path, so the data spends much of the time waiting in the send buffer. MPTCP's send buffer is effectively smaller because the large 3G RTT means it takes longer before DATA ACKs are returned to free space. If we manually reduce TCP's send buffer on the WiFi link, the latency can be reduced below that of MPTCP.

4.3 Coping with reordering

Most TCP implementations support Van Jacobson's fast path processing. The receiver assumes that data is received in-order and TCP quickly places the data received in-sequence in its receive buffer, either at the end of the in-order receive-queue (which the app can read) or at the end of the out-of-order queue. The latter happens when a packet was lost and we are waiting for the retransmission. In the rare case when a segment is received out of order, TCP scans the out-of-order queue to find the exact location of the received data.

With MPTCP the situation is completely different: while subflow sequence numbers are received in-order, data sequence numbers are often out-of-order forcing receivers to scan the large out-of-order queue. An obvious fix is to use a binary tree to reduce the out-of-order queue lookup time. This adds complexity to the code, and still takes logarithmic time to place a packet.

To obtain a simple, constant-time receive algorithm we leverage the way packets are sent: when a subflow is ready to send data, segments with contiguous data sequence numbers (a batch) will be allocated by the connection and sent on this subflow, as allowed by the subflow's congestion window. Each subflow, then, will receive in-order at the data level as long as the batch size is large. The receiver augments each subflow's data structures with a pointer to the connection-level out-of-order queue where it expects the next segment of that subflow to arrive. If the pointer is wrong, we revert to scanning the whole out-of-order queue.

This gives big benefits; the shortcuts work for 80% of the received packets. However, when the batch size is very small this optimization might not be enough. It is also not enough when we are receive-window limited and our retransmission mechanism kicks in.

For the 20% of the cases where the shortcut is not working, the receiver has to iterate over all packets in the out-of-order queue to insert the received packet. To avoid this behaviour, we modify the lookup mechanism as follows. First, the out-of-order queue groups in-sequence segments into batches. Then, we iterate over these batches instead of iterating over all the segments. As there are significantly less batches than packets in the out-of-order queue, the lookup process will be much faster.

We evaluate these algorithms by considering a client directly connected to a server by using two 1 Gbps links. The client starts a long download and we measure the receiver's CPU load. With more subflows the number of out-of-sequence segments that need to be processed increases; for clarity, we only present results with 2 subflows, a lower bound to utilize the links, and 8 subflows beyond which results are similar.

Figure 8 compares CPU load for the different receive algorithms. TCP (with 2 and 8 connections) is used as a benchmark. The *Tree* algorithm reduces CPU utilization, but *Shortcuts* and its improvement *AllShortcuts* help much more. When 8 subflows are used, CPU utilization drops from 42% to 30%, and when 2 subflows are used it drops from 25% to 20%.

5. MPTCP PERFORMANCE

The two main motivations to deploy MPTCP today are wireless networks where MPTCP could enable hosts to use both WiFi and 3G networks [20, 18] and datacenters where MPTCP allows servers to better exploit the load-balanced paths [19]. We experimentally evaluate the performance of our MPTCP implementation in these two environments.

5.1 MPTCP over WiFi and 3G

In the previous section, we used emulated networks to improve the algorithms used in our MPTCP implementation. Here, we use MPTCP over a 3G network offered

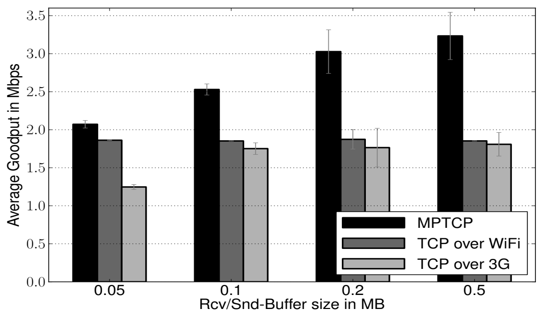


Figure 9: MPTCP used over real 3G and WiFi

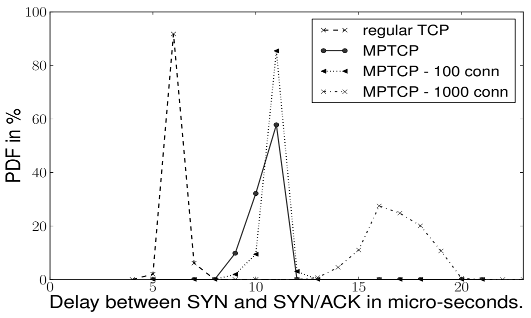


Figure 10: Connection establishment latency

by a commercial provider in Belgium; TCP's maximum throughput on this network is 2Mbps. Our MPTCP implementation correctly works over this 3G network despite its installed middleboxes. We also used a WiFi access point that was capped at 2 Mbps. This capping was implemented on the access point and would represent the bandwidth offered on a shared public WiFi network such as BT's FON. Figure 9 shows the average goodput achieved by TCP and MPTCP in function of the receive/send buffer sizes. Regular TCP achieves roughly the same goodput with both 3G and WiFi except when the buffer size is small where the larger round-trip-time penalizes the performance over 3G. MPTCP gets most of the available bandwidth when the buffer reaches 200KB, and it never underperforms TCP.

Our measurements show that MPTCP is able to utilize both the 3G and WiFi networks when the buffer is large enough. With a 500 KBytes buffer, MPTCP achieves almost the double of the goodput of regular TCP. With a 100 KBytes buffer, reaches a goodput that is 25% larger than regular TCP over WiFi or 3G.

5.2 Connection setup latency

During MPTCP connection setup the client and server generate a random key and verify that its hash is unique among all established connections (see Section 3.2). These keys are used later to verify the addition of new subflows. How does this affect connection setup latency?

The measurements use Xeon X5355 servers connected via Gigabit ethernet. Fig. 10 shows a PDF of the delay between receiving a SYN and sending the SYN/ACK, measured at the server. For regular TCP, 91% of the 20,000 connection setup attempts are processed in 6μs. Each connection is closed before the next attempt is made.

Setting up the first subflow of an MPTCP connection takes the server between 10 and 11μs if it has no established connections. The extra latency is mainly because MPTCP must hash the received key, generate the server key and verify that its hash is unique. If the server has established MPTCP connections, the verification of hash uniqueness is more expensive — Fig. 10 shows how the latency increases when the server already has 100 and 1000 established MPTCP connections.

This additional latency, although small compared to a LAN RTT, could be significantly reduced by maintaining a pool of precomputed keys.

5.3 HTTP performance

From the latency results, we can see that on a LAN, an MPTCP connection starts fractionally behind the equivalent TCP connection. A small amount of bandwidth and CPU cycles are also used to establish additional subflows. HTTP is notorious for generating many short connections. How long does an HTTP connection using MPTCP need to be for these startup costs to be outweighed by MPTCP's ability to use extra paths?

We directly connected a client and server via two gigabit links. For our tests we use `apachebench` [7], a benchmarking software developed by the Apache foundation that allows us to simulate a large number of clients interacting with an HTTP server. We configured `apachebench` to emulate 100 clients that generate 100000 requests for files of different sizes on a server (requests are closed-loop). The server was running MPTCP Linux and used `apache` version 2.2.16 with the default configuration.

We tested regular TCP that uses a single link, TCP with link-bonding using both interfaces and finally MPTCP. Fig. 11 shows the number of requests per second served in all three configurations. We expect MPTCP to be significantly better than regular TCP, and indeed this shows up in experiments: when the file sizes are larger than 100 KBytes MPTCP doubles the number of requests served. With files that are shorter than 30 KBytes, MPTCP decreases the performance compared to regular TCP. This is mainly due to the overhead of establishing and releasing a second subflow compared to the transmission time of a single file. These small flows take only a few RTTs and terminate while still in slowstart.

TCP with link-bonding performs very well especially when file sizes are small: the round-robin technique used

[7]http://httpd.apache.org/docs/2.0/programs/ab.html

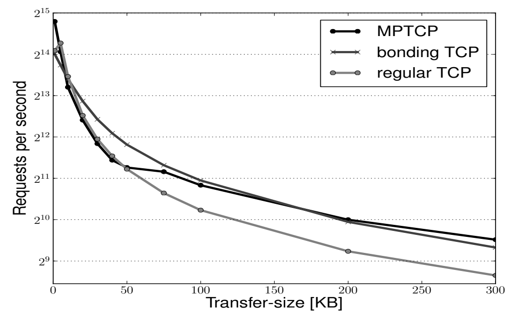


Figure 11: Apache-benchmark with 100 clients

by the Linux implementation manages to spread the load evenly, utilizing all the available capacity. MPTCP has a slight advantage over TCP with link-bonding only when file sizes are greater than 150KB in our experiment.

With larger files, there is a higher probability that link-bonding ends up congesting one of its two links, and some flows will be slower to finish. Flows on the faster link will finish quickly generating new requests, half of which will be allocated to the already congested link. This generates more congestion on an already congested link, with the effect that one link is highly congested while one is underutilized; the links will flip between the congested and underutilized states quasi randomly. We ran experiments with varying levels of congestion and found that MPTCP can serve 25% more requests than link bonding in such cases.

6. RELATED WORK

There has been a good deal of work on building multipath transport protocols[11, 24, 16, 10, 13, 4, 21, 5]. Most of this work aims to leave applications unchanged and focuses on the protocol mechanisms needed to implement multipath transmission. Key goals are robustness to long term path failures and to short term variations in conditions on the paths.

Huitema's Internet Draft [11] proposes using PCB identification to replace ports as demultiplexing points at the end-hosts; our connection tokens are similar in spirit. The proposal stripes segments over many addresses, using a single sequence number across all subflows.

Both MTCP [24] and M/TCP [21] use a single sequence number together with a scoreboard at the sender that allows maintaining congestion state and performing retransmissions per path. MTCP uses a single return path for ACKs which decreases its robustness; also it has been designed to run over an overlay network (RON), reducing its deployability and efficiency.

pTCP [10] is one of the most complete proposals to date. The SYN exchange signals the addresses that will be used in the multipath connection, and this set is fixed - pTCP does not support mobility. Congestion control and retransmissions are performed per subflow. At connection level there are global send and receive buffers; a data sequence number and acknowledgment helps deal with reordered data at the connection level. This is inserted in a pTCP header that follows the TCP header.

RMTP [16] is a rate-based protocol targeted for mobile hosts that uses packet-pairs to estimate available bandwidth on each path and supports both reliable an unreliable delivery. RMTP does not offer the same service as TCP, and requires app changes.

The Stream Control Transmission Protocol (SCTP) has been designed with multihoming in mind to support telephony signaling applications. SCTP's multihoming support was initially only capable of recovering from failures. However, several authors have extended it to support load-sharing [2, 13]. SCTP-CMT [12] uses a single sequence number across all paths and keeps a scoreboard and other information to have accurate per-path congestion windows and to drive retransmissions.

Our protocol design has drawn on all this literature, and has been further guided by our experimental study of middleboxes. In light of this study, none of the existing approaches are deployable as most use single subflow sequence numbers which will be dropped. pTCP does not use subflow sequence numbers but it is unclear how its additional headers should be encoded. Further, pTCP will not cope with resegmenting or content-changing middleboxes.

On the OS side, none of the previous works on TCP have addressed the practical problems of getting multipath transport working in reality. Most use simulation analysis, and do not consider receive-buffer issues. SCTP-CMT has been implemented in the FreeBSD kernel, but its performance has not been evaluated in detail.

A technique that was previously proposed to reduce the size of the receive-buffer is to use sender-side scheduling to get the packets "in-order" at the receiver (see e.g. [17]). Unfortunately, this solution is brittle: any packet losses or just variations in RTT will disrupt the ordering, causing the receiver to buffer just as much data. Further, the sender still has to buffer as much data as before.

7. LESSONS LEARNED

In today's Internet, the three-way-handshake involves not only the two communicating hosts, but also all the middleboxes on the path. Verifying the presence of a particular TCP option in a SYN+ACK is not sufficient to ensure that a TCP extension can be safely used. As shown in [9], some middleboxes pass TCP options that they don't understand. This is safe for TCP options that are purely informative (e.g. RFC1323 timestamps) but causes problems with other options such as those that redefine the semantics of TCP header fields. For example, the large window extension in RFC1323 changes the semantics of the window field of the TCP header and extends it beyond 16 bits. Nearly 20 years after the publication of RFC1323, there are still stateful firewalls that do not understand this option in SYNs but block data packets that are sent in the RFC1323 extended window. A TCP extension that changes the semantics of parts of the packet header must include mechanisms to cope with middleboxes that do not understand the new semantics.

In an end-to-end Internet, all the information carried inside TCP packets is immutable. Today this is no longer true: the entire TCP header and the payload must be considered as mutable fields. If a TCP extension needs to rely on a particular field, it must check its value in a way that cannot be circumvented by middleboxes that do not understand this extension. The DSM checksum is an example of a solution to deal with these problems.

Most importantly, deployable TCP extensions must necessarily include techniques that enable them to fall-back to regular TCP when something wrong happens. If a middlebox interferes badly with a TCP extension, the problem must be detected and the extension automatically disabled to preserve the data transfer. A TCP extension will only be deployed if it guarantees that it will transfer data correctly (and hopefully better) in all the cases where a regular TCP is able to transfer data.

8. CONCLUSIONS

TCP was designed when the Internet strictly obeyed the end-to-end principle and each host had a single IP address. Single-homing is disappearing and a growing fraction of hosts have multiple interfaces/addresses. In this paper we evaluated whether TCP can be extended to efficiently support such hosts.

We explored whether it was possible to design Multipath TCP in a way that is still deployable in today's Internet. The answer is positive, but any major change to TCP must take into account the various types of middleboxes that have proliferated. In fact, they influenced most of the design choices in Multipath TCP besides the congestion control. Our experiments show that MPTCP safely operates through all the middleboxes we've identified in our previous study [9].

From an implementation viewpoint, we proposed new algorithms to solve practical but important problems such as sharing a limited receive buffer between multiple flows on a smartphone, or optimizing the MPTCP receive code. Experiments show that our techniques are effective, making MPTCP ready for adoption.

This work highlights once again the fact that hidden middleboxes increase the complexity of the Internet, making evolution difficult. We should revisit the Internet architecture to recognize explicitly their role. The big challenge, however, is to build a solution that is deployable in today's Internet.

Acknowledgments

We would like to thank all the members of the Trilogy project and of the MPTCP IETF working-group who have helped us to shape the MPTCP protocol. Xinxing Hu wrote the first version of the Click elements that model middlebox behaviours.

This work has been supported partially by Google through a University Research project and by FP7 projects ECODE and CHANGE. Christoph Paasch has partially been supported by Nokia Research.

9. REFERENCES

[1] S. Barré. *Implementation and Assessment of Modern Host-based Multipath Solutions*. PhD thesis, Université catholique de Louvain, November 2011.

[2] M. Becke et al. Load sharing for the stream control transmission protocol (sctp). Internet draft, draft-tuexen-tsvwg-sctp-multipath-02.txt, work in progress, July 2011.

[3] A. Bittau, M. Hamburg, M. Handley, D. Mazières, and D. Boneh. The case for ubiquitous transport-level encryption. In *USENIX Security'10*, pages 26–26, Berkeley, CA, USA, 2010. USENIX Association.

[4] Y. Dong, D. Wang, N. Pissinou, and J. Wang. Multi-path load balancing in transport layer. In *EuroNGI Conference*, 2007.

[5] A. Ford, C. Raiciu, M. Handley, and O. Bonaventure. TCP extensions for multipath operation with multiple addresses, Jan 2012. IETF draft (work in progress).

[6] FreeBSD Project. tcp – internet transmission control protocol. *FreeBSD Kernel Interfaces Manual*.

[7] M. Handley. Why the internet only just works. *BT Technology Journal*, 24:119–129, 2006.

[8] M. Handley, V. Paxson, and C. Kreibich. Network intrusion detection: evasion, traffic normalization, and end-to-end protocol semantics. In *Proc. USENIX Security Symposium*, pages 9–9, 2001.

[9] M. Honda, Y. Nishida, C. Raiciu, A. Greenhalgh, M. Handley, and H. Tokuda. Is it still possible to extend TCP? In *IMC 2011, 11th Internet Measurement Conference*, Nov. 2011.

[10] H.-Y. Hsieh and R. Sivakumar. A transport layer approach for achieving aggregate bandwidths on multi-homed mobile hosts. In *Proc. MobiCom '02*, pages 83–94, New York, NY, USA, 2002. ACM.

[11] C. Huitema. Multi-homed TCP. Internet draft, IETF, 1995.

[12] J. Iyengar, P. Amer, and R. Stewart. Performance implications of a bounded receive buffer in concurrent multipath transfer. *Computer Communications*, 30(4), February 2007.

[13] J. R. Iyengar, P. D. Amer, and R. Stewart. Concurrent multipath transfer using SCTP multihoming over independent end-to-end paths. *IEEE/ACM Trans. Netw.*, 14(5):951–964, 2006.

[14] P. Key, L. Massoulié, and D. Towsley. Path selection and multipath congestion control. In *Proc. IEEE Infocom*, May 2007.

[15] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The click modular router. *ACM Trans. Comput. Syst.*, 18:263–297, August 2000.

[16] L. Magalhaes and R. Kravets. Transport level mechanisms for bandwidth aggregation on mobile hosts. *ICNP*, page 0165, 2001.

[17] F. Mirani, M. Kherraz, and N. Boukhatem. Forward prediction scheduling: Implementation and performance evaluation. In *ICT 2011*, pages 321 –326, may 2011.

[18] C. Pluntke, L. Eggert, and N. Kiukkonen. Saving mobile device energy with Multipath TCP. In *MobiArch*, 2011.

[19] C. Raiciu, S. Barré, C. Pluntke, A. Greenhalgh, D. Wischik, and M. Handley. Improving datacenter performance and robustness with Multipath TCP. In *Proc ACM Sigcomm*, 2011.

[20] C. Raiciu, D. Niculescu, M. Bagnulo, and M. Handley. Opportunistic mobility with Multipath TCP. In *MobiArch*, 2011.

[21] K. Rojviboonchai and H. Aida. An evaluation of multi-path transmission control protocol (M/TCP) with robust acknowledgement schemes. *IEICE Trans. Communications*, 2004.

[22] P. Srisuresh and M. Holdrege. IP Network Address Translator (NAT) terminology and considerations. RFC 2663, August 1999.

[23] D. Wischik, C. Raiciu, A. Greenhalgh, and M. Handley. Design, implementation and evaluation of congestion control for multipath tcp. In *NSDI'11*, Berkeley, CA, USA, 2011. USENIX Association.

[24] M. Zhang, J. Lai, A. Krishnamurthy, L. Peterson, and R. Wang. A transport layer approach for improving end-to-end performance and robustness using redundant paths. In *Proc USENIX '04*, 2004.

The TCP Outcast Problem: Exposing Unfairness in Data Center Networks

Pawan Prakash, Advait Dixit, Y. Charlie Hu, Ramana Kompella
Purdue University
{pprakash, dixit0, ychu, rkompella}@purdue.edu

Abstract

In this paper, we observe that bandwidth sharing via TCP in commodity data center networks organized in multi-rooted tree topologies can lead to severe unfairness, which we term as the *TCP Outcast problem*, under many common traffic patterns. When many flows and a few flows arrive at two ports of a switch destined to one common output port, the small set of flows lose out on their throughput share significantly (almost by an order of magnitude sometimes). The Outcast problem occurs mainly in taildrop queues that commodity switches use. Using careful analysis, we discover that taildrop queues exhibit a phenomenon known as *port blackout*, where a series of packets from one port are dropped. Port blackout affects the fewer flows more significantly, as they lose more consecutive packets leading to TCP timeouts. In this paper, we show the existence of this TCP Outcast problem using a data center network testbed using real hardware under different scenarios. We then evaluate different solutions such as RED, SFQ, TCP pacing, and a new solution called *equal-length routing* to mitigate the Outcast problem.

1 Introduction

In recent years, data centers have emerged as the cornerstones of modern communication and computing infrastructure. Large-scale online services are routinely hosted in several large corporate data centers comprising upwards of 100s of thousands of machines. Similarly, with cloud computing paradigm gaining more traction, many enterprises have begun moving some of their applications to the public cloud platforms (*e.g.*, Amazon EC2) hosted in large data centers. In order to take advantage of the economies of scale, these data centers typically host many different classes of applications that are independently owned and operated by completely different entities—either different customers or different divisions within the same organization.

While resources such as CPU and memory are strictly sliced across these different tenants, network resources are still largely shared in a *laissez-faire* manner, with TCP flows competing against each other for their fair share of the bandwidth. Ideally, TCP should achieve *true fairness* (also known as RTT fairness in the literature), where each flow obtains equal share of the bottleneck link bandwidth. However, given TCP was designed to achieve long-term throughput fairness in the Internet, today's data center networks inherit TCP's *RTT bias*, *i.e.*, when different flows with different RTTs share a given bottleneck link, TCP's throughput is inversely proportional to the RTT [20]. Hence, low-RTT flows will get a higher share of the bandwidth than high-RTT flows.

In this paper, we observe that in many common data center traffic scenarios, even the conservative notion of fairness with the RTT bias does not hold true. In particular, we make the surprising observation that in a multi-rooted tree topology, when a large number of flows and and a small set of flows arrive at different input ports of a switch and destined to a common output port, the small set of flows obtain significantly lower throughput than the large set. This observation, which we term as the *TCP Outcast problem*, is surprising since when it happens in data center networks, the small set of flows typically have lower RTTs than the large set of flows and hence, according to conventional wisdom, should achieve higher throughput. Instead, we observe an *inverse RTT bias*, where low-RTT flows receive much lower throughput and become 'outcast'ed from the high-RTT ones.

The TCP Outcast problem occurs when two conditions are met: First, the network comprises of commodity switches that employ the simple taildrop queuing discipline. This condition is easily met as today's data center networks typically use low- to mid-end commodity switches employing taildrop queues at lower levels of the network hierarchy. Second, a large set of flows and a small set of flows arriving at two different input ports compete for a bottleneck output port at a switch. Interestingly, this condition also happens often in data center networks due to the nature of multi-rooted tree topologies and many-to-one traffic patterns of popular data center applications such as MapReduce [8] and Partition/Aggregate [3]. In particular, from any receiver node's perspective, the number of sender nodes that are $2n$ routing hops (n to go up the tree and n to come down) away grows exponentially, (*e.g.*, in a simple binary tree, the number grows as $2^n - 2^{n-1}$). Thus, if we assume we can place map/reduce tasks at arbitrary nodes in the data center, it is likely to have disproportionate numbers of incoming flows to different input ports of switches near the receiver node. When the above two conditions are met, we observe that the flows in the smaller set end up receiving much lower per-flow throughput than the flows in the other set—almost an *order of magnitude* smaller in many cases. We observed this effect in both real testbeds comprising commodity hardware switches and in simulations.

The reason for the existence of the TCP Outcast problem can be attributed mainly to a phenomenon we call *port blackout* that occurs in taildrop queues. Typically, when a burst of packets arrive at two ports that are both draining to an output port, the taildrop queuing discipline leads to a short-term blackout where one of the ports loses a series of incoming packets compared to the other. This behavior can affect either of the ports; there is no significant bias against any of them. Now, if one of the ports consists of a few flows, while the other has many flows (see Section 2), the series of packet drops affect the set of few flows since each of them can potentially lose the tail of an entire congestion window resulting in a timeout. TCP timeouts are quite catastrophic since the TCP sender will start the window back from one and it takes a long time to grow the congestion window back.

It is hard not to observe parallels with the other well-documented problem with TCP in data center networks, known as the TCP Incast problem [23]. The Incast problem was observed first in the context of storage networks where a request to a disk block led to several storage servers connected to the same top-of-rack (ToR) switch to send a synchronized burst of packets, overflowing the limited buffer typically found in commodity data center switches causing packet loss and TCP timeouts. But the key problem there was under-utilization of the capacity since it took a long time for a given TCP connection to recover as the default retransmission timeout was rather large. In contrast, the TCP Outcast problem exposes unfairness and, unlike Incast, it requires neither competing flows to be synchronized, nor the bottleneck to be at the ToR switch. One major implication of the Outcast problem, similar to the Incast problem, is the need to design protocols that minimize the usage of switch packet buffers. Designing protocols that take advantage of additional information in the data center setting to reduce buffer consumption in the common case can result in a range of benefits, including reducing the impact of Outcast. DCTCP [3] is one such effort but there may be related efforts (*e.g.*, RCP [9]) that may also be beneficial.

One key question that remains is why one needs to worry about the unfairness across flows within the data center. There are several reasons for this: (1) In a multi-tenant cloud environment with no per-entity slicing of network resources, some customers may gain unfair advantage while other customers may get poor performance even though both pay the same price to access the network. (2) Even within a customer's slice, unfairness can affect the customer's applications significantly: In the reduce phase of map-reduce applications (*e.g.*, sort), a reduce node fetches data from many map tasks and combines the partial results (*e.g.*, using merge sort). If some connections are slower than the others, the progress of the reduce task is stalled, resulting in significantly increased memory requirement in addition to slowing down the overall application progress. (3) TCP is still the most basic light-weight solution that provides some form of fairness in a shared network fabric. If this solution itself is broken, almost all existing assumptions about any level of fairness in the network are in serious jeopardy.

Our *main contribution* in this paper is to show the existence of the TCP Outcast problem under many different traffic patterns, with different numbers of senders and bottleneck locations, different switch buffer sizes, and different TCP variants such as MP-TCP [21] and TCP Cubic. We carefully isolate the main reason for the existence of the TCP Outcast problem using simulations as well as with traces collected at an intermediate switch of a testbed. We further investigate a set of practical solutions that can deal with the Outcast problem. First, we evaluate two router-based approaches—stochastic fair queuing (SFQ) that solves this problem to a large extent, and RED, which still provides only conservative notion of TCP fairness, *i.e.*, with RTT bias. Second, we evaluate an end-host based approach, TCP pacing, and show that pacing can help reduce but does not eliminate the Outcast problem completely.

TCP's congestion control was originally designed for the "wild" Internet environment where flows exhibiting a diverse range of RTTs may compete at congested links. As such, the RTT bias in TCP is considered a reasonable compromise between the level of fairness and design complexity and stability. In contrast, data centers present a tightly maintained and easily regulated environment which makes it feasible to expect a stricter notion of fairness, *i.e.*, true fairness. First, many network topologies (*e.g.*, multi-rooted trees, VL2 [13]) exhibit certain symmetry, which limits the flows to a small number of possible distances. Second, newly proposed topologies such as fat-trees that achieve full bisection bandwidth make shortest-path routing a less stringent requirement.

Motivated by the above reasoning, we propose and evaluate a simple new routing technique called *equal-length routing* that essentially side-steps shortest-path routing and makes all paths equal length. This simple counter-intuitive approach promotes better mixing of traffic reducing the impact of port blackouts. The technique effectively achieves true fairness, *i.e.*, equal throughput for competing flows sharing a congested link anywhere in the network since the flow RTTs are also balanced. The obvious downside to this approach is that it results in wasting resources near the core of the network. For certain topologies such as the fat-tree that already provide full bisection bandwidth, this may be alright since capacity is anyway provisioned. For data center networks with over-subscription, this approach will not be suitable; it may be better to employ techniques such as SFQ queuing if true fairness is desired.

2 Unfairness in Data Center Networks

In this section, we first provide a brief overview of today's data center networks. We then discuss our main observation in today's commodity data center networks, namely the TCP Outcast problem, which relates to unfair sharing of available link capacity across different TCP flows.

2.1 Data Center Network Design

The key goal of any data center network is to provide rich connectivity between servers so that networked applications can run efficiently. For full flexibility, it is desirable to build a data center network that can achieve full bisection bandwidth, so that any server can talk to any server at the full line rate. A lot of recent research (*e.g.*, fat-tree [2], VL2 [13]) has focused on building such full bisection bandwidth data center networks out of commodity switches. Most practical data center topologies are largely in the form of multi-rooted multi-level trees, where servers form the leaves of the tree are connected through switches at various levels—top-of-rack (ToR) switches at level 1, aggregation switches at level 2 and finally, core switches at level 3. Such topologies provide the necessary rich connectivity by providing several paths with plenty of bandwidth between server pairs.

Table 1: List of some 48-port COTS switches

48-port Switches	Congestion Avoidance
HP/3Com E5500G	Taildrop
Juniper EX4200	Taildrop
Brocade FastIron GS series	Taildrop
Cisco Catalyst 3750-E	Weighted Taildrop
Cisco Nexus 5000	Taildrop

Data center networks today are largely built out of commodity off-the-shelf (COTS) switches, primarily to keep the costs low. While these switches offer full line-rate switching capabilities, several features found in high-end routers are often missing. In particular, they typically have shallow packet buffers and contain small forwarding tables among other such deficiencies. In addition, they also typically implement simple queueing disciplines such as taildrop. In Table 1, we can see that almost all the commodity switches that are produced by popular vendors employ the taildrop (or variants of taildrop) queue management policy.

The choice transport protocol in most data centers today is TCP, mainly because it is a three-decade old mature protocol that is generally well-understood by systems practitioners and developers. Two aspects of TCP are generally taken for granted: First, TCP utilizes the network as effectively as possible, and hence is *work-conserving*; if there is spare bandwidth in the network, TCP will try to utilize it. Second, TCP is a *fair* protocol; if there are multiple flows traversing a bottleneck link, they share the available bottleneck capacity in a fair manner. These two aspects of TCP typically hold true in both wide-area as well as local-area networks.

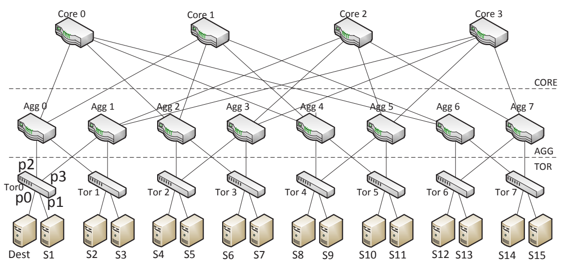


Figure 1: Data Center Topology

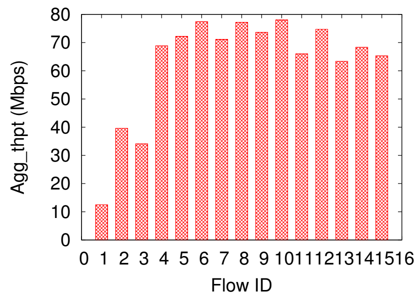
Unfortunately, certain data center applications create pathological conditions for TCP, causing these seemingly taken-for-granted aspects of TCP to fail. In cluster file systems [5, 25], for instance, clients send parallel reads to dozens of nodes, and all replies need to arrive before the client can proceed further—exhibiting a barrier-synchronized many-to-one communication pattern. When synchronized senders send data in parallel in a high-bandwidth low-latency network, the switch buffers can quickly overflow, leading to a series of packet drops which cause TCP senders to go into the timeout phase. Even when the capacity opens up, still some senders are stuck for a long time in the timeout phase, causing severe underutilization of the link capacity. This observation is famously termed as the *TCP Incast problem* first coined by Nagle *et al.* in [18]. The Incast problem has ever since generated a lot of interest from researchers—to study and understand the problem in greater depth [7, 27] as well as propose solutions to alleviate it [23, 26].

In this paper, we focus on a different problem that relates to the second aspect of TCP that is taken for granted, namely, TCP fairness.

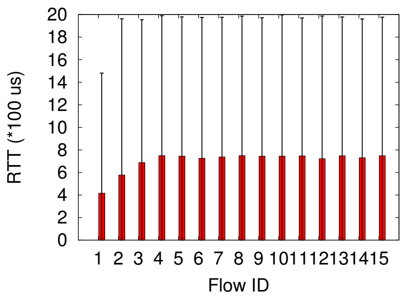
2.2 The TCP Outcast Problem

Consider a data center network that is organized in the form of an example (k=4) fat-tree topology as shown in Figure 1 proposed in literature [2]. (While we use the fat-tree topology here for illustration, this problem is not specific to fat-trees and is found in other topologies as well.) Recall that in a fat-tree topology, all links are of the same capacity (assume 1Gbps in this case). Now suppose there are 15 TCP flows, f_i ($i = 1...15$) from sender S_i to $Dest$. In this case, the bottleneck link is the last-hop link from $ToR0$ to $Dest$. All these flows need not start simultaneously, but we mainly consider the portion of time when all the 15 flows are active.

We built a prototype data center network using NetFPGA-based 4-port switches (discussed in more de-



(a) Aggregate throughput



(b) Average RTT

Figure 2: TCP Outcast problem

tail in Section 3.1) and experimented with the traffic pattern discussed above. Figure 2 shows that the per-flow aggregate throughput obtained by these 15 flows exhibit a form of somewhat surprising unfairness: Flow f_1 which has the shortest RTT achieves significantly lesser aggregate throughput than any of f_4-f_{15}—almost 7-8$\times$ lower. Flows f_2 and f_3 also achieve lesser throughput (about 2$\times$) than f_4-f_{15}. We observe this general trend under many different traffic patterns, and at different locations of the bottleneck link (discussed in detail in Section 3.2), although the degree of unfairness varies across scenarios. (Note that we also found evidence of unfairness by conducting a limited number of experiments with another institution's data center testbed consisting of commodity HP-ProCurve switches organized in a $k = 4$ fat-tree topology.)

We call this gross unfairness in the throughput achieved by different flows that share a given bottleneck link in data center networks employing commodity switches with the taildrop policy as the *TCP Outcast problem*. Here, flow f_1, and to some extent f_2 and f_3, are 'outcast'ed by the swarm of other flows $f_4 - f_{15}$. Although $f_1 - f_3$ differ in their distances (in terms of number of hops between the sender and the receiver) compared to flows $f_4 - f_{15}$, this does not alone explain the Outcast problem. If any, since f_1 has a short distance of only 2 links, its RTT should be much smaller than $f_4 - f_{15}$ which traverse 6 links. This is confirmed in Figure 2(b), which shows the average RTT over time along with the max/min values. According to TCP analysis [20], throughput is inversely proportional to the RTT, which suggests f_1 should obtain higher throughput than any of $f_4 - f15$. However, the Outcast problem exhibits exactly the opposite behavior.

The reason for this counter-intuitive result is two fold: First, taildrop queuing leads to an occasional *"port blackout"* where a series of back-to-back incoming packets to one port are dropped. Note that we deliberately use the term blackout to differentiate from a different phenomenon called 'lockout' that researchers have associated with taildrop queues in the past [10, 6]. The well-known lockout problem results from global synchronization of many TCP senders, where several senders transmit synchronized bursts, and flows that manage to transmit ever so slightly ahead of the rest manage to get their packets through but not the others, leading to unfairness. In contrast, the blackout problem we allude to in this paper occurs when two input ports drain into one output port, with both input ports containing a burst of back-to-back packets. In this case, one of the ports may get lucky while the other may incur a series of packet drops, leading to a temporary blackout for that port. Second, if one of the input ports contains fewer flows than the other, the temporary port blackout has a catastrophic impact on that flow, since an entire tail of the congestion window could be lost, leading to TCP timeouts. We conduct a detailed analysis of the queueing behavior to elaborate on these reasons in Section 4.

2.3 Conditions for TCP Outcast

To summarize, the two conditions for the Outcast problem to occur are as follows:

- (C1) The network consist of COTS switches that use the taildrop queue management discipline.
- (C2) A large set of flows and a small set of flows arriving at two different input ports compete for a bottleneck output port at a switch.

Unfortunately, today's data centers create a perfect storm for the Outcast problem to happen. First, as we mentioned before, for cost reasons, most data center networks use COTS switches (Table 1) which use the taildrop queue management discipline, which exhibits the port blackout behavior.

Second, it is not at all uncommon to have a large set of flows and a small set of flows arriving at different input ports of a switch and compete for a common output port, due to the nature of multi-rooted tree topologies commonly seen in data center networks and typical traffic patterns in popular data center applications such as MapReduce [8] and Partition/Aggregate [3]. For instance, in large MapReduce applications, the many map and reduce tasks are assigned to workers that span a large portion of the data center network. When a reduce task initiates multiple TCP connections to different map tasks, the sources of the flows are likely to reside in different parts of the network. As a result, in any tree-like topology, it is very likely that these flow sources result in disproportionate numbers of flows arriving at different input ports of a switch near the receiver. This is because from any leaf node's perspective, the number of nodes that are $2n$ hops away grows exponentially. For example, in a simple binary tree, the number grows as $2^n - 2^{n-1}$.

3 Characterizing Unfairness

In this section, we present experimental results that demonstrate the throughput unfairness symptom of the TCP Outcast problem. We extensively vary all the relevant parameters (*e.g.*, traffic pattern, TCP parameters, buffer size, queueing model) to extensively characterize the TCP Outcast problem under different conditions.

3.1 Experimental Setup

Our data center testbed is configured in a $k = 4$ fat-tree as shown in Figure 1, with 16 servers at the leaf-level and 20 servers acting as the switches. The servers are running CentOS 5.5 with Linux kernel 2.6.18. Each switch server is equipped with NetFPGA boards acting as a 4-port Gigabit switch and running OpenFlow for controlling the routing. Each NetFPGA card has a packet buffer of 16KB per port, and all the queues in the NetFPGA switches use taildrop queuing.

Packets are routed in the fat-tree by statically configuring the shortest route to every destination. When there are multiple shortest paths to a destination, the static route at each switch is configured based on the trailing bits of the destination address. For example, at ToR switches, a packet destined to a server connected to a different ToR switch is forwarded to one of the aggregate switches as decided by the last bit of the destination address: the right aggregate switch if the bit is 0, and the left if the bit is 1. Similarly, at the aggregate switch, packets coming from ToR switches that are destined to a different pod are forwarded to one of the core switches. The aggregate switch selects the core switch based on the second last bit of the destination address. Our routing scheme is in principle similar to the Portland architecture [19].

To emulate condition C2 in Section 2.3, we used a simple many-to-one traffic pattern, mainly for convenience, for most of our experiments. We also used a more general pattern (Section 3.2.4) to show many-to-one pattern is not a necessity.

The many-to-one traffic pattern naturally leads to sources placed at different distances. For instance, consider the fat-tree topology in Figure 1 (same is true with other multi-rooted tree topologies such as VL2). From the perspective of any receiver, the senders belong to 3 classes—senders under the same ToR (2-hop), senders in the same pod (4-hop), and senders in different pods (6-hop). The senders belonging to a particular class are at the same distance from the receiver. We conducted experiments under different scenarios with 15 senders ($S1$-$S15$) as depicted in Figure 1, each of which initiates one or more TCP flows to a single receiver (labeled $Dest$), and measured the TCP throughput share achieved by different flows. Note that in all experiments, unless otherwise noted, we only consider the throughput share obtained by individual flows when *all* flows are active.

We used the default values for all TCP parameters except the minimum round-trip timeout (minRTO), which we set to 2 milliseconds to overcome the adverse effects of TCP Incast problem [23]. We disabled TCP segmentation offload since that would have further increased the burstiness of TCP traffic and probably increased unfairness. We experimented with different TCP variants (Reno, NewReno, BIC, CUBIC) and obtained similar results. Due to page limit, we present results under TCP BIC and CUBIC for experiments conducted on our testbed and under NewReno for ns-2 simulations.

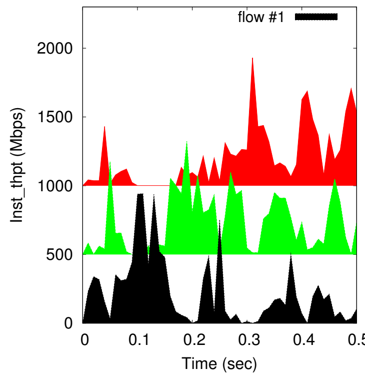
3.2 Throughput Unfairness Results

We start with the simplest base case where one flow from each sender is initiated to the destination, and show that it results in the TCP Outcast problem. We then show that the problem exists even if (1) there is disparity in the number of competing flows arriving at different input ports; (2) flows do not start simultaneously, unlike in the incast [26] problem; (3) the bottleneck link is in the core of the network; and (4) there is background traffic sharing the bottleneck link.

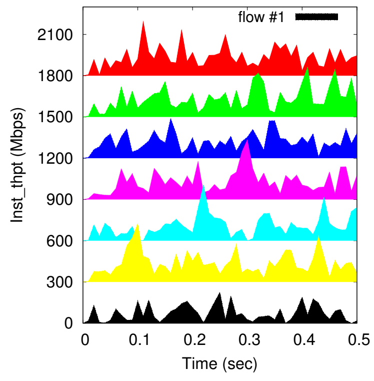
3.2.1 Case I – Different Flow Proportions

In this case, multiple long-distance flows, one from each sender node six hops away, arriving at port $p2$ of $ToR0$, and one flow from sender $S1$ (flow 1), arriving at port $p1$ of $ToR0$, compete for output port $p0$. Figures 3(a)-3(c) show the instantaneous throughput achieved by individual flows within the first 0.5 seconds when there are two, six, and twelve 6-hop flows, respectively. The y-axis for each flow is offset by 500, 300, and 150 Mbps respectively so that the instantaneous throughput per flow is clearly visible. Figures 3(d)-3(f) show the corresponding aggregate throughput of flow 1 and the average aggregated throughput of all the 6-hop flows within the first 0.1, 0.2, and 0.5 seconds.

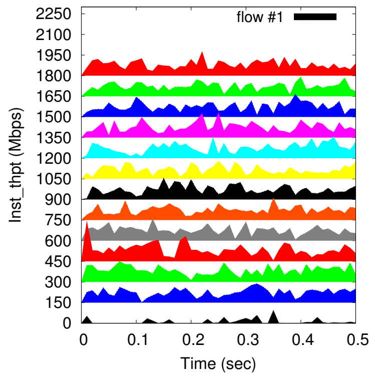
These figures show that with two 6-hop flows, flow 1 gets higher than average share in the beginning, but is occasionally starved by other flows after 0.1 seconds (due to reasons explained in Section 4). Figures 3(a)-3(b) show that as we increase the number of long-distance flows to six and twelve, flow 1's throughput becomes increasingly worse and practically starves compared to the



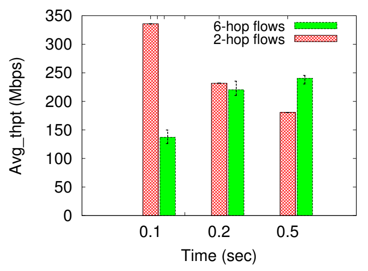
(a) Two 6-hop flows



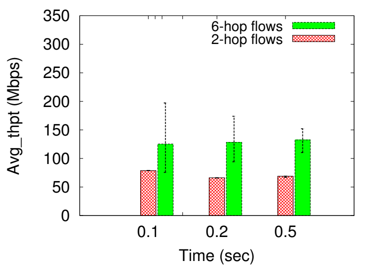
(b) Six 6-hop flows



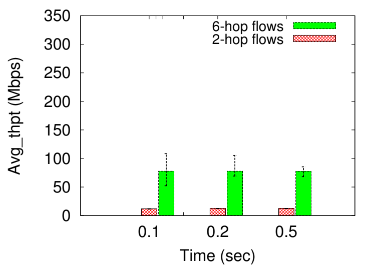
(c) Twelve 6-hop flows



(d) Two 6-hop flows

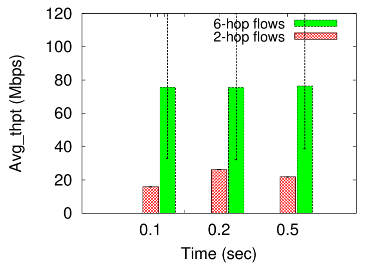


(e) Six 6-hop flows

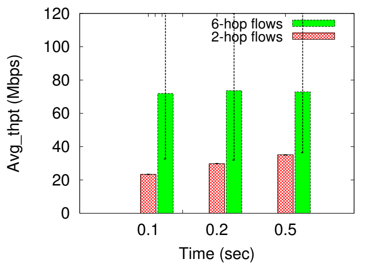


(f) Twelve 6-hop flows

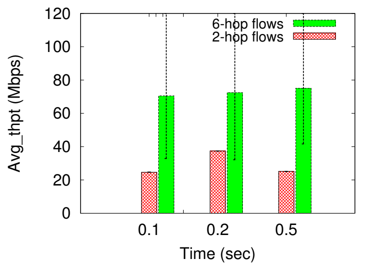
Figure 3: Instantaneous and average throughput in case of one 2-hop flow and multiple 6-hop flows



(a) 10 2-hop, 120 6-hop



(b) 20 2-hop, 240 6-hop



(c) 30 2-hop, 360 6-hop

Figure 4: Average throughput per host in case of fixed proportion (1:12) of 2-hop and 6-hop flows

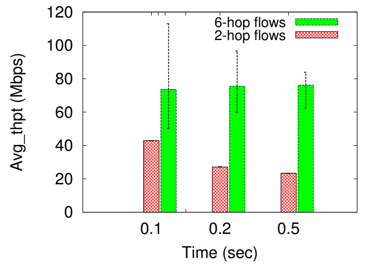
rest. Overall, the aggregate throughput of flow 1 is 2× and 7× worse than the average throughput of the six and twelve 6-hop flows, respectively.

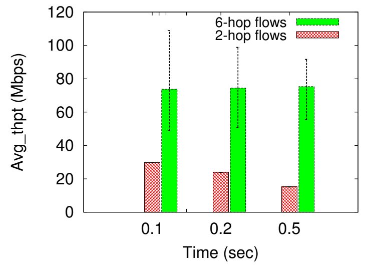
3.2.2 Case II – Multiple Flows

Figure 3(f) shows the throughput unfairness when one 2-hop flow at one input port competes with twelve 6-hop flows at the other input port for access to the output port at switch $ToR0$. To explore whether the problem persists even with a larger numbers of flows competing, we vary the number of flows per host (10, 20 and 30) while keeping the ratio of flows arriving at the two input ports the same (1:12) as in Figure 3(f). Figure 4 shows when 10 flows arrive at port $p1$ and 120 flows arrive at port $p2$, the average throughput of the 2-hop flows is 3× worse than that of the 120 6-hop flows. Note that the y-axis in the figure is the average throughput on a per-host basis (*i.e.*, sum of all flows that originate at the host); since same number of flows start at all the nodes, the individual per-flow throughput is just scaled down by the appropriate number of flows per host. We see that the similar unfairness persists even in this scenario.

3.2.3 Case III – Unsynchronized Flows

One could perhaps conjecture that throughput unfairness observed so far may be because all flows are starting at the same time, similar to the TCP Incast problem. In order to verify whether this is a requirement, we experiment with staggered flow arrivals. We again consider the same thirteen flows (one 2-hop and twelve 6-hop flows) as in Figure 3(f), but instead of starting all flows at the same time, we stagger the start time of the 2-hop flow to be 100ms before, 100ms after, and 200ms after the 6-hop flows. In Figure 5, we can observe that the 2-


(a) Early start of 100ms for 2-hop flow


(b) Delayed start of 100ms for 2-hop flow

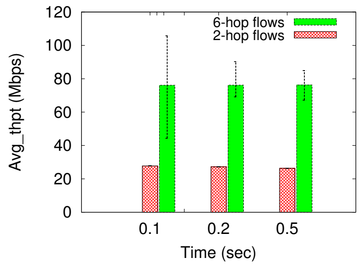
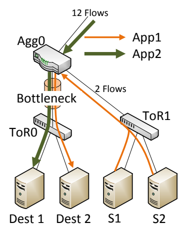

(c) Delayed start of 200ms for 2-hop flow

Figure 5: Average throughput in case of different start times for the 2-hop flow. Time on x-axis is relative to when all flows are active.



(a) Scenario

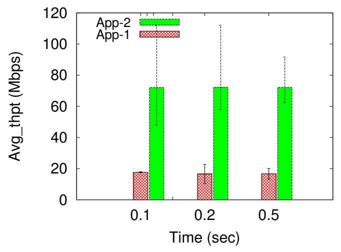

(b) Throughput

Figure 6: Two applications sharing a network bottleneck

hop flow obtains significantly lower throughput even if it starts 100ms before the 6-hop flows, which allows it sufficient time to ramp up to a larger window. It appears that once the 6-hop flows begin, the 2-hop flows experience starvation as in Figure 3(f). In Figure 5(a), we can clearly observe the throughput disparity between the 2-hop flow and 6-hop flows increases with time as the impact of the initial gains due to initial higher window reduces over time.

3.2.4 Case IV – Distinct Flow End-points

So far, we have mainly experimented with the many-to-one traffic pattern. In order to show that this is not fundamental to TCP Outcast, we use a different traffic pattern where the 2-hop and 6-hop flows have different end-hosts, as show in 6(a). Here, the bottleneck link lies between the aggregate switch and the ToR switch, which is again different from TCP Incast. Figure 6(b) shows that unfairness persists, confirming that TCP Outcast can happen in aggregate switches and does not rely on many-to-one communication. Further, as shown in the previous section, these flows need not be synchronized. Together, these non-requirements significantly increase the likelihood of observing the TCP Outcast problem in production data center environments.

3.2.5 Case V – Background Traffic

Since many different applications may share the network fabric in data centers, in this experiment, we study if background traffic sharing the bottleneck switch can eliminate or at least mitigate the Outcast problem. We generated background traffic at each node similar to the experiments in [13], by injecting flows between random pairs of servers that follow a probability distribution of flow sizes (inferred from [13]). The network bandwidth consumed by the background traffic is controlled by the flow inter-arrival time. Specifically, if we want B background traffic, given a mean flow size F, the mean flow inter-arrival time is set as F/B, and we create an exponential distribution of flow inter-arrival time with the calculated mean. In this experiment, we also generated two 4-hop flows to confirm that there is nothing specific about 2-hop and 6-hop flows contending.

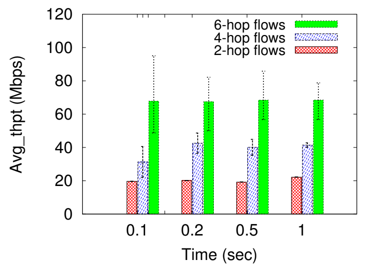
Figure 7 depicts the resulting average throughput for one 2-hop flow, two 4-hop flows, and twelve 6-hop flows under different amounts of background traffic. Clearly the presence of background traffic affects the average throughput of every flow. But the extent of unfairness is not mitigated by the background traffic completely. In particular, the gap between the throughput of 2-hop flows and 6-hop flows remain 4× and 2.5× under background traffic of 10%, 20% of the bottleneck link capacity (1 Gbps) respectively. Only when the background traffic reaches 50% of the bottleneck link capacity, the unfairness seems to taper off, that too after 0.2 seconds.

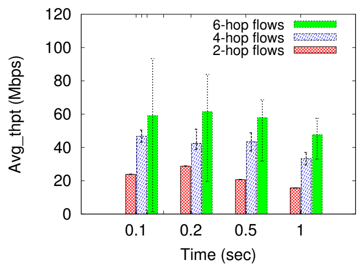
3.2.6 Case VI – Other Experiment Scenarios

We also vary other parameters such as buffer sizes and RTT on the TCP Outcast problem.

Buffer size. We found that increasing the buffer size does not have a significant effect on the TCP Outcast problem. A larger buffer size means that it would take longer for the queue to fill up and for port blackout to happen but it eventually happens. In our testbed, we have tried with buffer sizes of 16KB and 512KB and found that the unfairness still persists. Using ns-2 simulations, we simulated different buffer sizes of 32, 64, 128KB, and found similar results.

RTT. We simulate twelve flows from one 4-hop server and one flow from the other 4-hop sender (hence all flows have the same RTTs). We observed that the TCP Out-


(a) 100 Mbps background traffic


(b) 200 Mbps background traffic

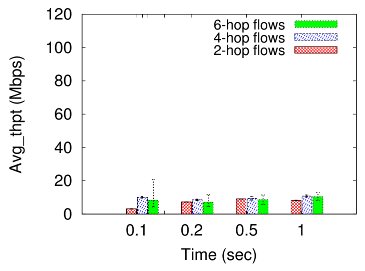
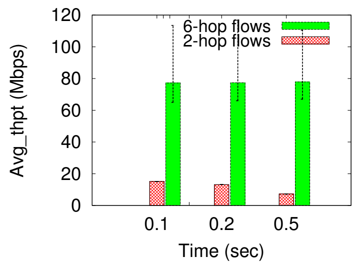
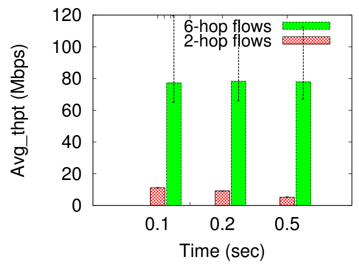

(c) 500 Mbps background traffic

Figure 7: Average throughput of 2-hop, 4-hop, and 6-hop flows under background traffic


(a) TCP CUBIC without SACK


(b) TCP CUBIC with SACK

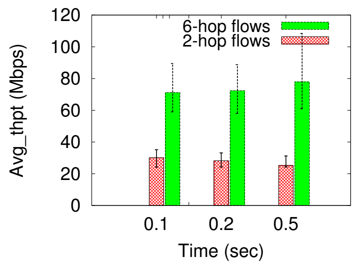

(c) MPTCP

Figure 8: Average throughput of 2-hop, and 6-hop flows under CUBIC, SACK, and MPTCP.

cast problem still exists and the single flow is starved for throughput. Thus, it appears the number of flows on the ports, as opposed to the RTT differential, impacts the unfairness.

Multiple ports contending. In the test bed, we modified routing so that we have flows coming on 3 input ports and going to one output port. Even in this case, the TCP Outcast problem is present. In ns-2, we have experimented with even more input ports (*e.g.*, 6-pod, 8-pod fat-tree, VL2 [13] topology) and found that the Outcast problem exists.

CUBIC, SACK, and MPTCP. We tested the existence of Outcast problem with TCP CUBIC with and without SACK, and with MPTCP [21]. Figure 8 depicts the occurence of Outcast in all these scenarios, although MPTCP seems to reduce the extent of unfairness. Since MPTCP opens many different sub-flows corresponding to each TCP flow, this scenario is roughly equivalent to the multiple flows experiment in Figure 4.

4 Explaining Unfairness

Routers with taildrop queues have been known to suffer from the *lockout* problem, in which a set of flows experience regular packet drops while other flows do not. Floyd *et al.* [10] have demonstrated that TCP phase effects can lead to these lockouts where packets arriving at a router after certain RTTs find the queue to be full and hence are dropped. TCP phase effects were studied in the context of the Internet and RTT was the primary factor in determining which flows will suffer from lockout.

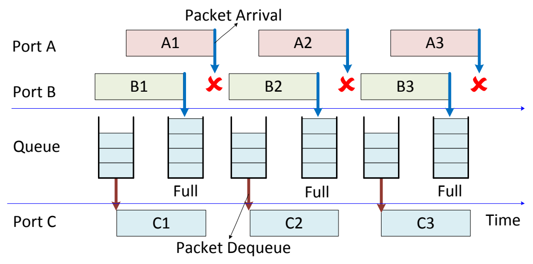
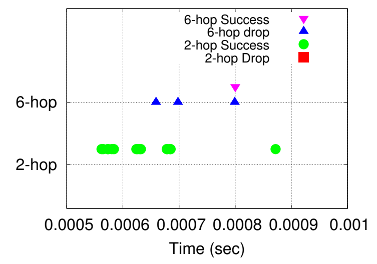


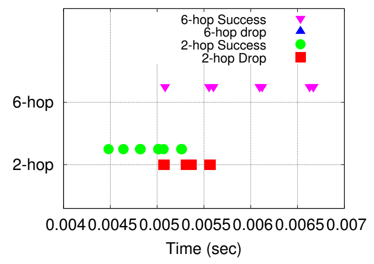
Figure 9: Timeline of port blackout

In this section, we demonstrate the existence of a different phenomenon called *port blackout* in the context of data center networks. Port blackout is defined as the phenomenon where a stream of back-to-back packets arriving on multiple input ports of a switch compete for the same output port, and packets arriving on one of the input ports are dropped while packets arriving on the other input ports are queued successfully in the output port queue. Port blackouts occurs when the switch uses taildrop queue management policy.

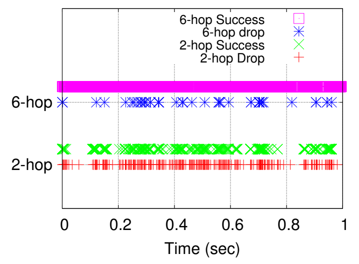
In the following, we explain how port blackouts can occur in data center networks. We also corroborate our observation with ns-2 simulations with configurations identical to our testbed. We then introduce a drop model using ns-2 simulation to demonstrate the effects of port blackout on TCP throughput. We end with insights into how port blackout can be prevented in data center networks.



(a) Blackout for 6-hop flows



(b) Blackout for 2-hop flows



(c) Packet loss and success event for 2-hop and 6-hop flows

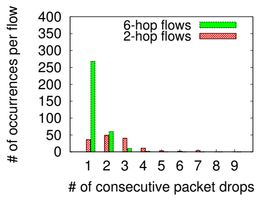
Figure 10: Blackout behavior observed in taildrop queues

4.1 Port Blackout in Data Center Testbed

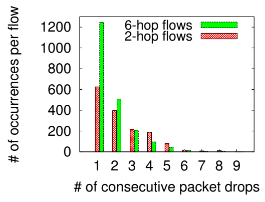
Figure 9 schematically depicts the timeline of events occurring at a switch amidst a port blackout episode. A stream of packets $A1$, $A2$ and $A3$ arriving at port A and $B1$, $B2$ and $B3$ arriving at port B are competing for output port C which is full. Since most of the competing flows are long flows, their packets are of the same size, which means the time spent by each of the frames is the same on the wire. Now, since these packets arrive on two different ports, they are unlikely arriving at exactly the same time (the ports are clocked separately). However, the inter-frame spacing on the wire is the same for both ports, since there are back-to-back packets (assuming the senders are transmitting many packets) and no contention from any other source on the Ethernet cable (given switched Ethernet). Now, due to the asynchronous nature of these packet arrivals, one port may have packets slightly ahead of the others, *e.g.* in Figure 9, port B's packets arrive just slightly ahead of port A's packets.

After de-queuing a packet $C1$, the output port queue size drops to $Q-1$. Now, since packets from B arrive slightly ahead of A, packet $B1$ arrives at port B next (denoted by an arrow on the time line), finds queue size to be $Q-1$, and is successfully enqueued in the output queue making it full. Next, packet $A1$ arrives at port A, finds the queue to be full, and hence gets dropped. The above pattern of consecutive events then repeats, and $A2$ as well as $A3$ end up with the same fate as its predecessor $A1$. This synchronized chain of events among the three ports can persist for some time resulting in a sequence of packet losses from one input port, *i.e.*, that port suffers a blackout. Once the timing is distorted, either because there is a momentary gap in the sending pattern or due to some other randomness in timing, this blackout may stop. But, every so often, one of the ports may enter into this blackout phase, losing a bunch of consecutive packets. We note that either of the input ports can experience this blackout phenomenon; there is no intrinsic bias against any one port.

To investigate the existence of port blackout in our data center testbed, we collected the traffic traces close to output port $p0$ and input ports $p1$ and $p2$ of switch



(a) With 1 flow per host



(b) With 20 flows per host

Figure 11: Distribution of consecutive packet drops

$ToR0$ during the experiment in Figure 3(f). The trace at port $p1$ consists of a stream of back-to-back packets from server $S1$ which is under the same $ToR0$. The trace at port $p2$ consists of packets from servers that are located outside $ToR0$. Both these streams of packets are meant to be forwarded toward output port $p0$ and hence, compete with each other. Correlating these two traces with the traffic trace at output port $p0$, we can infer the set of packets that were successfully forwarded and the set that were dropped.

Figure 10(c) shows the timeline of packets successfully sent and dropped at port $p2$ (for 6-hop flows) and port p1 (for 2-hop flows) of switch $ToR0$ during the experiment. When port blackouts happen, we can observe clusters of packet drops. To see the detailed timing of packet events during blackouts, we zoom into small time intervals. Figure 10(a) depicts a small time interval (about 500 microseconds) when port $p2$ carrying the flows from servers outside $ToR0$ experiences a port blackout, during which packets from port $p1$ are successfully sent while consecutive packets from port $p2$ are dropped. Figure 10(b) depicts a similar blackout event for port $p1$. While we highlight a single incident of port blackout here, Figure 11(a) shows the distribution of episodes with k consecutive packet losses. As we can see, the 2-hop flow experiences many more episodes of 3 and 4 consecutive packet drops than the 6-hop flows. This trend does not seem to change even with a larger number of flows per host as shown in Figure 11(b).

4.2 Port Blackout Demonstration in ns-2

While the traces above give us some insight that black-

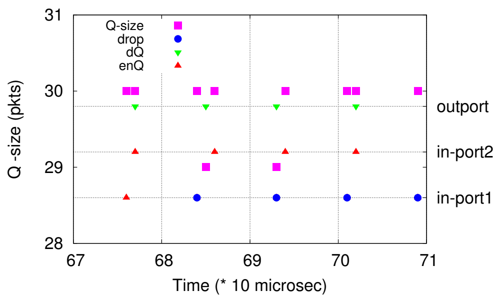


Figure 12: Queue dynamics at the ToR 0 switch

outs may be happening, due to inaccuracies in timing we only get rough insights from the above trace. In order to understand this even more closely, we resort to simulations in ns-2. In simulations, we can easily observe the changing value of the dynamic output port queue size and the exact timing of how it correlates with packet enqueue and dequeue events. We simulate the same experiment, *i.e.*, fat-tree configuration and traffic pattern, as in Figure 3(c), in ns-2. Figure 12 depicts the exact timeline of different packet events, enqueue, dequeue, and drop, corresponding to the three ports.

For each port (right y-axis), packet enqueue, drop, and dequeue events are marked. The left y-axis shows the queue size at any given instant of time. The queue dynamics is shown for one of the intervals during which in-port1 is suffering from a port blackout episode. Consider the interval between 68 and 69 (*10 microsecond). First, a packet arrives at in-port1. But the queue was full (Q-size 30) at that instant as denoted by the square point. As a result this packet at in-port1 is dropped by the taildrop policy. Soon after that, a packet is dequeued from the output queue and now the queue size drops to 29. Next, a new packet arrives at in-port2, and is accepted in the queue making the queue full again. This pattern repeats and in-port1 suffers from consecutive packet drops, leading to an episode of port blackout.

4.3 Effect of Port Blackout on Throughput

We have explained the root cause for the port blackout phenomenon in previous sections using real traces collected from our data center testbed as well as using the ns-2 simulations. In this section, we present a simulation model to help us understand the impact of port blackout on the throughput of TCP flows. More specifically, we want to analyze the relationship between the number of flows on an input port (that experiences blackout) and the impact on their TCP throughput due to port blackout.

We simulate a simple topology in ns-2 consisting of a single sender node (node 1) and a single receiver node (node 2) connected via a switch. To simulate the port blackout behavior, we modified the taildrop queue at the switch to operate in two states. In the ON state, it drops all packets that it receives from node 1. In the OFF state,

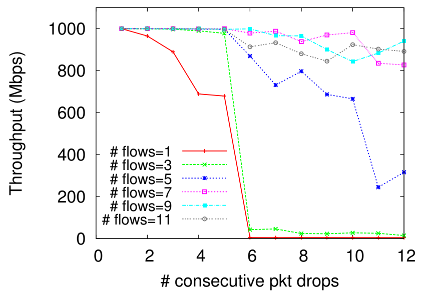


Figure 13: Effect of consecutive packet drops on TCP

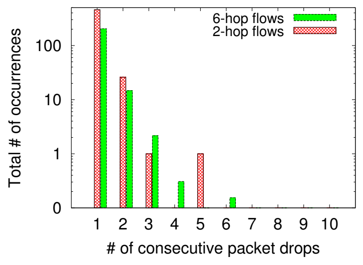
it does not drop any packet. The queue toggles from OFF to ON state after every k seconds, where k is chosen from an exponential distribution with a mean of 0.005 seconds, which is the approximate time period between two blackout periods we observed in our testbed. It remains in ON state for a fixed duration that corresponds to m consecutive packet drops. Note that an ON state does not necessarily correspond to m actual packet drops; it is the *time duration* in which the switch would have dropped m consecutive packets. In other words, we only drop consecutive packets if they appear back-to-back during the ON state.

Using this drop model, we study the impact of the length of ON duration on the throughput of f TCP flows from node 1. Figure 13 shows the aggregate TCP throughput (on y-axis) of f flows, as the number of consecutive packet drops m (on x-axis) varies. We observe that when there are 7 or more flows, port blackout, *i.e.* consecutive packets drops during the ON state, only affects the throughput of the flows slightly, even as m grows to 10. This is because packets dropped in the ON state are spread across the flows and each flow can recover quickly from few packet losses due to fast retransmission. However, when there are few flows, the consecutive packet drops have a catastrophic effect on their throughput because of timeouts that leads to reducing the congestion window significantly.

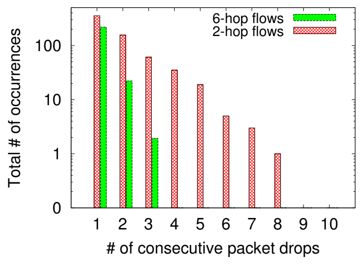
While it may appear from the above experiment that the Outcast problem may disappear if we have larger number of flows, Figure 4 clearly indicates that that is not true. The reason lies in the fact that if there are a larger number of flows, the duration of the blackout simply increases causing more consecutive packet drops, translating to a similar number of packet losses per flow as before. We find evidence of this effect in Figure 11(b) which shows the number of consecutive drops for 2-hop flows remains much higher than the 6-hop flows even with 20 flows per host.

5 Mitigating Unfairness

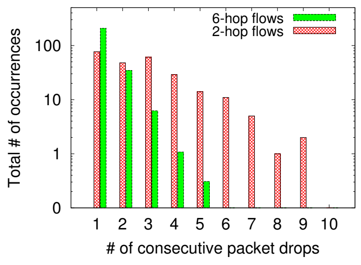
The root cause of the TCP Outcast problem in data center networks is input port blackout at bottleneck switches



(a) RED

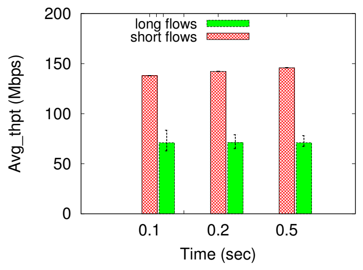


(b) SFQ

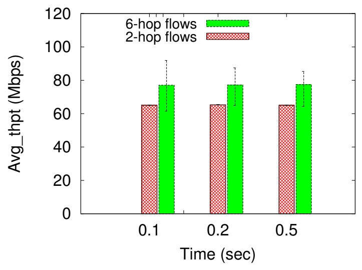


(c) TCP pacing

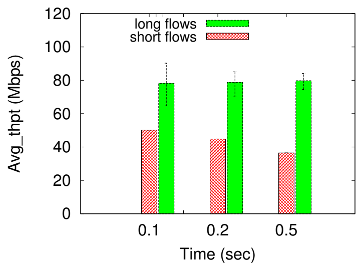
Figure 14: Distribution of consecutive packet drops under RED, SFQ, and TCP pacing solutions



(a) RED



(b) SFQ



(c) TCP pacing

Figure 15: Average throughput under RED, SFQ, and TCP pacing solutions

happening due to the taildrop policy of the output queue, which has a drastic effect on the throughput of the few flows that share the blackout input port. Hence, the key to solving the TCP Outcast problem is to distribute packet drops among all competing flows (for an output port) arriving at the switch to avoid blackout of any flow.

In this section, we study three approaches that all achieve this goal, but via rather different means. The first includes two solutions that directly get rid of the taildrop packet drop policy, by replacing it with RED or SFQ. The second, TCP pacing, tries to alleviate the burstiness of packets in each TCP flow (*i.e.*, window), and hence potentially reduces bursty packet loss for any particular flow. The third approach avoids port blackout by forcing flows with nearby senders to detour to take similar paths as flows with faraway senders so that their packets are well interleaved along the routing paths. We evaluate the effectiveness of each approach and further discuss their pros and cons in terms of implementation complexity and feasibility in data center networks.

5.1 RED

RED [11] is an active queue management policy which detects incipient congestion and randomly marks packets to avoid window synchronization. The random marking of packets essentially interleaves the packets from different input ports to be dropped and hence avoids blackout of any particular port. We simulate RED in ns-2 with the same configuration as Figure 3(f), with 12 6-hop flows and 1 2-hop flow destined to a given receiver. In our setup, we use the classical RED policy, with the minimum threshold set to 5 packets, the maximum threshold set to 15 packets, and the queue weight set to 0.002.

Figure 14(a) shows the distribution of different number of consecutive packet drops for 2-hop and 6-hop flows (since there are multiple 6-hop flows we take an average of all the twelve flows). We observe that the consecutive packet drop events are similar for 2-hop and 6-hop flows. More than 90% of packet drop events consist of a single consecutive packet loss, suggesting that blackouts are relatively uncommon, and all the flows should have achieved a fair share of TCP throughput. However, Figure 15(a) shows a difference in average throughput between 2-hop and 6-hop flows. This is explained by the well-known RTT bias that TCP exhibits; since the 2-hop flow has a lower RTT, it gets the a larger share of the throughput (TCP throughput $\sim \frac{1}{RTT \times \sqrt{droprate}}$). Thus, we can clearly see that RED queuing discipline achieves RTT bias but does not provide the true throughput fairness in data center networks.

5.2 Stochastic Fair Queuing

We next consider stochastic fair queuing (SFQ) [17], which was introduced to provide fair share of throughput to all the flows arriving at a switch irrespective of their RTTs. It divides an output buffer into buckets (the number of buckets is a tunable parameter) and the flows sharing a bucket get their share of throughput corresponding to the bucket size. A flow can also opportunistically gain a larger share of the bandwidth if some other flow is not utilizing its allocated resources. We simulate the same experimental setup as before (twelve 6-hop and one 2-hop flow) in ns-2 with SFQ packet scheduling. We set the number of buckets to 4 to simulate the common case where there are fewer buckets than flows.

Figures 15(b) shows the average throughput observed by different flows. We see that SFQ achieves almost equal throughput (true fairness) between the 6-hop flows and the 2-hop flow. We can also observe in Figure 14(b) that the 2-hop flow experiences a higher percentage of consecutive packet drop events (20% of the time, it experiences 2 consecutive drops). Since the 2-hop flow has a lower RTT, it is more aggressive as compared to the 6-hop flows, leading to more dropped packets than those flows.

5.3 TCP Pacing

TCP pacing, also known as "packet spacing", is a technique that spreads the transmission of TCP segments across the entire duration of the estimated RTT instead of having a burst at the reception of acknowledgments from the TCP receiver (*e.g.*, [1]). Intuitively, TCP pacing promotes the interleaving of packets of the TCP flows that compete for the output port in the TCP Outcast problem and hence can potentially alleviate blackout on one input port. We used the TCP pacing in our ns-2 [24] setup and repeated the same experiment as before. Figure 15(c) shows that TCP pacing reduces throughput unfairness; the throughput gap between the 2-hop flow and 6-hop flows is reduced from 7× (Figure 3(f)) to 2×. However, the Outcast problem remains. This is also seen in Figure 14(c), where the 2-hop flow still experiences many consecutive packet drops. The reason is as follows. There is only a single (2-hop) flow arriving at one of the input ports of the bottleneck switch. Hence, there is a limit on how much TCP pacing can space out the packets for that flow, *i.e.* the RTT of that 2-hop flow divided by the congestion window.

5.4 Equal-Length Routing

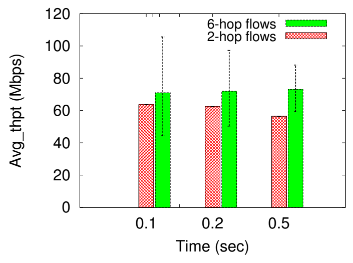
As discussed in Section 3, one of the conditions for TCP Outcast problem is the asymmetrical location of senders of different distances to the receiver, which results in disproportionate numbers of flows on different input ports of the bottleneck switch competing for the same output port. Given we can not change the location of the servers, one intuitive way to negate the above condition is to make flows from all senders travel similar paths and hence their packets are well mixed in the shared links and hence well balanced between different input ports. Before discussing how to achieve this, we briefly discuss a property of the fat-tree network topology that makes the proposed scheme practical.

In a fat-tree [2] topology, each switch has the same amount of fan-in and fan-out bandwidth capacity, and hence the network is fully provisioned to carry the traffic from the lowest level of servers to the topmost core switches and vice versa. Thus although the conventional shortest path routing may provide a shorter RTT for packets that do not need to reach the top-most core switches, the spare capacity in the core cannot be used by other flows anyways.

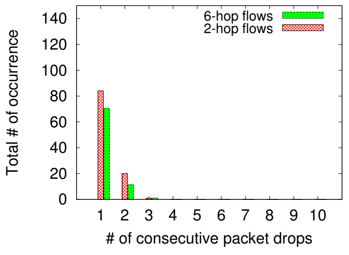
Based on the above observation, we propose *Equal-length routing* in a fat-tree data-center network topology, where data packets from every server are forwarded up to the core switch irrespective of whether the destination belongs in the same pod of the sender. Effectively, Equal-length routing prevents the precarious situations where a given flow alone suffers from consecutive packet losses (as discussed in Section 3). Since all the flows are routed to the core of the network, there is enough mixing of the traffic arriving at various ports of a core switch that the packet losses are uniformly shared by multiple flows. Equal-length routing ensures a fair share among multiple flows without conceding any loss to the total network capacity. It is simple to implement and requires no changes to the TCP stack.

Implementation. Equal-length routing can be implemented in a fat-tree by routing each packet to a core switch randomly or deterministically chosen. Under the random scheme, the core switch is randomly uniformly chosen [13]. Under the deterministic scheme, the core switch is determined based on the destination address as follows. On our testbed running OpenFlow for routing control, at the ToR switches, a packet coming from a server (down port) is forwarded to one of the aggregate switches (up ports) as decided by the destination address (*e.g.*, port selection is based on the last bit of the destination address). Similarly at the aggregate switches, packets coming from ToR (down) are forwarded to the core (up) switches and vice versa (*e.g.*, port selection based on the second last bit of the destination address). Consider the flow of packets from $S1$ to the $Dest$ in Figure 1. Without the Equal-length routing, the packets take the path $S1 \rightarrow Tor0 \rightarrow Dest$, but under Equal-length routing, the packets will go through $S1 \rightarrow Tor0 \rightarrow Agg0 \rightarrow Core0 \rightarrow Agg0 \rightarrow Tor0 \rightarrow Dest$.

Properties. Equal-length routing creates interesting changes in the dynamics of interactions among the TCP flows. Under the new routing scheme, all the flows are mixed at core switches (feasible in a network providing full-bisection bandwidth) which gives rise to two properties: (1) The deterministic scheme results in all flows in many-to-one communication sharing the same downward path, whereas the random scheme results in flows



(a) Average throughput



(b) Consecutive packet drops

Figure 16: Effectiveness of equal-length routing

going to the same destination being well balanced between different input ports at each switch in the downward paths. Both effects avoid the blackout of any particular flow; instead, all the competing flows suffer uniform packet losses. (2) All the flows have similar RTTs and similar congestion window increases. Together, they ensure that competing flows achieve similar true fair share of the bottleneck link bandwidth.

Evaluation. To analyze the proposed routing scheme, we implemented Equal-length routing in our data center testbed and conducted similar experiments as Figure 3(f). Other than the new routing scheme, all the setup was kept the same. We analyze the TCP throughput achieved by different flows as before. Note that even though every flow is now communicating via a core switch, we label them as 2-hop and 6-hop flows for consistency and ease of comparison with previous results.

Figure 16(a) depicts the TCP throughput share between different flows. We can observe that the flows get a fair throughput share which is comparable to what they achieved under the SFQ packet scheduling discipline. The fair share of throughput can be further explained from Figure 16(b), which shows that the flows experience similar packet drops; none of the flows has to suffer a large number of consecutive packet drops.

5.5 Summary

Table 2 summarizes the four potential solutions for the TCP Outcast problem we have evaluated. All solutions share the common theme of trying to break the synchronization of packet arrivals by better interleaving packets of the flows competing for the same output port and

Techniques	Fairness Property
RED	RTT bias
SFQ	RTT fairness
TCP Pacing	Inverse RTT bias
Equal-length routing	RTT fairness

Table 2: Fairness property of TCP Outcast solutions

hence evening out packet drops across them. We find that although all approaches alleviate the TCP outcast problem, RED still leads to RTT bias, and TCP pacing still leads to significant inverse RTT bias. SFQ and Equal-length routing provide RTT fairness but have their limitations too. SFQ is not commonly available in commodity switches due to its complexity and hence overhead in maintaining multiple buckets, and Equal-length routing is feasible only in network topologies without oversubscription. The final choice of solution will depend on the fairness requirement, traffic pattern, and topology of the data center networks.

6 Related Work

We divide related work into three main categories—TCP problems in data centers, new abstractions for network isolation/slicing, and TCP issues in the Internet context.

TCP issues in data centers. Much recent work has focused on exposing various problems associated with TCP in data centers (already discussed before in Section 2). The TCP Incast problem was first exposed in [18], later explored in [23, 7, 26]. Here the authors discover the adverse impact of barrier-synchronized workloads in storage network on TCP performance. [23] proposes several solutions to mitigate this problem in the form of fine-grained kernel timers and randomized timeouts, *etc*.

In [3], the authors observe that TCP does not perform well in mixed workloads that require low latency as well as sustained throughput. To address this problem, they propose a new transport protocol called DC-TCP that leverages the explicit congestion notification (ECN) feature in the switches to provide multi-bit feedback to end hosts. While we have not experimented with DC-TCP in this paper, the Outcast problem may potentially be mitigated since DC-TCP tries to ensure that the queues do not become full. We plan to investigate this as part of our future work.

In virtualized data centers, researchers have observed serious negative impact of virtual machine (VM) consolidation on TCP performance [15, 12]. They observe that VM consolidation can slow down the TCP connection progress due to the additional VM scheduling latencies. They propose hypervisor-based techniques to mitigate these negative effects.

In [21], the authors propose multipath TCP (MPTCP) to improve the network performance by taking advantage of multiple parallel paths between a given source and a destination routinely found in data center environments.

MPTCP does not eliminate the Outcast problem as we discussed in Section 3.2.6.

Network isolation. The second relevant body of work advocates network isolation and provides each tenant with a fixed share of network resources [14, 22, 4]. For example, SecondNet [14] uses rate controllers in hypervisors to ensure per-flow rate limits. Seawall [22] uses hypervisors to share the network resources according to some pre-allocated weight to each customer. Finally, Oktopus [4] provides a virtual cluster and a two-tier oversubscribed cluster abstraction, and also uses hypervisors to implement these guarantees. Our focus in this paper, however, is on the flow-level fairness as opposed to tenant-level isolation considered in these solutions.

Wide-area TCP issues. While this paper is mainly in the context of data centers, several TCP issues have been studied for almost three decades in the wide-area context. One of the most related work is that by Floyd *et al.* [10], where they study so-called phase effects on TCP performance. They discover that taildrop gateways with strongly periodic traffic can result in systematic discrimination and lockout behavior against some connections. While our port blackout phenomenon occurs because of systematic biases long mentioned in this classic work and others (*e.g.*, RFC 2309 [6]), they do not mention the exact Outcast problem we observe in this paper. RTT bias has also been documented in [10] where TCP throughput is inversely proportional to the RTT. TCP variants such as TCP Libra [16] have been proposed to overcome such biases, but are generally not popular in the wild due to their complexity. The typical symptom of the TCP Outcast problem in data centers is the exact opposite.

7 Conclusion

The quest for fairness in sharing network resources is an age-old one. While the fairness achieved by TCP is generally deemed acceptable in the wide-area Internet context, data centers present a new frontier where it may be important to reconsider TCP fairness issues. In this paper, we present a surprising observation we call the TCP Outcast problem, where if many and few flows arrive at two input ports going towards one output port, the fewer flows obtain much lower share of the bandwidth than the many flows. Careful investigation of the root cause revealed the underlying phenomenon of port blackout where each input port occassionally loses a sequence of packets. If these consecutive drops are distributed over a small number of flows, their throughput can reduce significantly because TCP may enter into the timeout phase. We evaluate a set of solutions such as RED, SFQ, TCP pacing, and a new solution called Equal-length routing that can mitigate the Outcast problem. In ongoing work, we are investigating the effect of the Outcast problem on real applications such as MapReduce.

8 Acknowledgements

The authors thank Praveen Yalagandula for help with experiments on HP's testbed. The authors also thank Jitu Padhye, Amin Vahdat (our shepherd) and anonymous reviewers for various comments that helped improve the paper. This work was supported in part by NSF Awards CNS-1054788 and CRI-0751153.

References

[1] A. Aggarwal, S. Savage, and T. E. Anderson. Understanding the Performance of TCP Pacing. In *IEEE INFOCOM*, 2000.

[2] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. In *ACM SIGCOMM*, 2008.

[3] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data Center TCP (DCTCP). In *ACM SIGCOMM*, 2010.

[4] H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron. Towards Predictable Datacenter Networks. In *ACM SIGCOMM*, 2011.

[5] P. J. Braam. File systems for clusters from a protocol perspective. `http://www.lustre.org`.

[6] B. Braden, D. Clark, J. Crowcroft, et al. Recommendations on queue management and congestion avoidance in the internet. RFC 2309, 1998.

[7] Y. Chen, R. Griffith, J. Liu, R. H. Katz, and A. D. Joseph. Understanding TCP Incast Throughput Collapse in Datacenter Networks. In *USENIX WREN*, 2009.

[8] J. Dean, S. Ghemawat, and G. Inc. MapReduce: Simplified Data Processing on Large Clusters. In *USENIX OSDI*, 2004.

[9] N. Dukkipati, M. Kobayashi, R. Zhang-Shen, and N. McKeown. Processor sharing flows in the internet. In *IEEE IWQoS*, June 2005.

[10] S. Floyd and V. Jacobson. On traffic phase effects in packet-switched gateways. *Internetworking: Research and Experience*, 1992.

[11] S. Floyd and V. Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM Trans. Netw.*, August 1993.

[12] S. Gamage, A. Kangarlou, R. R. Kompella, and D. Xu. Opportunistic Flooding to Improve TCP Transmit Performance in Virtualized Clouds. In *2nd ACM Symposium on Cloud Computing (SOCC'11)*, 2011.

[13] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. VL2: A Scalable and Flexible Data Center Network. In *ACM SIGCOMM*, 2009.

[14] C. Guo, G. Lu, H. J. Wang, S. Yang, C. Kong, P. Sun, W. Wu, and Y. Zhang. Secondnet: a data center network virtualization architecture with bandwidth guarantees. In *ACM CoNEXT*, 2010.

[15] A. Kangarlou, S. Gamage, R. R. Kompella, and D. Xu. vSnoop: Improving TCP Throughput in Virtualized Environments via Acknowledgement Offload. In *ACM/IEEE SC*, 2010.

[16] G. Marfia, C. Palazzi, G. Pau, M. Gerla, M. Y. Sanadidi, and M. Roccetti. TCP-Libra: Exploring RTT Fairness for TCP. Technical report, UCLA Computer Science Department, 2005.

[17] P. E. McKenney. Stochastic fairness queueing. In *IEEE INFOCOM*, pages 733–740, 1990.

[18] D. Nagle, D. Serenyi, and A. Matthews. The panasas activescale storage cluster: Delivering scalable high bandwidth storage. In *ACM/IEEE SC*, 2004.

[19] R. Niranjan Mysore, A. Pamboris, N. Farrington, N. Huang, P. Miri, S. Radhakrishnan, V. Subramanya, and A. Vahdat. Portland: a scalable fault-tolerant layer 2 data center network fabric. In *ACM SIGCOMM*, 2009.

[20] J. Padhye, V. Firoiu, D. Towsley, and J. Kurose. Modeling TCP Throughput: A Simple Model and its Empirical Validation. In *ACM SIGCOMM*, 1998.

[21] C. Raiciu, S. Barre, C. Pluntke, A. Greenhalgh, D. Wischik, and M. Handley. Improving datacenter performance and robustness with multipath tcp. In *ACM SIGCOMM*, 2011.

[22] A. Shieh, S. Kandula, A. Greenberg, and C. Kim. Seawall: Performance Isolation for Cloud Datacenter Networks. In *USENIX HotCloud*, 2010.

[23] V. Vasudevan, A. Phanishayee, H. Shah, E. Krevat, D. G. Andersen, G. R. Ganger, G. A. Gibson, and B. Mueller. Safe and effective fine-grained TCP retransmissions for datacenter communication. In *ACM SIGCOMM*, 2009.

[24] D. X. Wei. *A TCP pacing implementation for NS2*, April 2006 (accessed October 03, 2011).

[25] B. Welch, M. Unangst, Z. Abbasi, G. Gibson, B. Mueller, J. Small, J. Zelenka, and B. Zhou. Scalable performance of the panasas parallel file system. In *6th USENIX Conference on File and Storage Technologies (FAST)*, 2008.

[26] H. Wu, Z. Feng, C. Guo, and Y. Zhang. ICTCP: Incast Congestion Control for TCP in data center networks. In *ACM CoNEXT*, 2010.

[27] J. Zhang, F. Ren, and C. Lin. Modeling and Understanding TCP Incast in Data Center Networks. In *IEEE INFOCOM*, 2011.