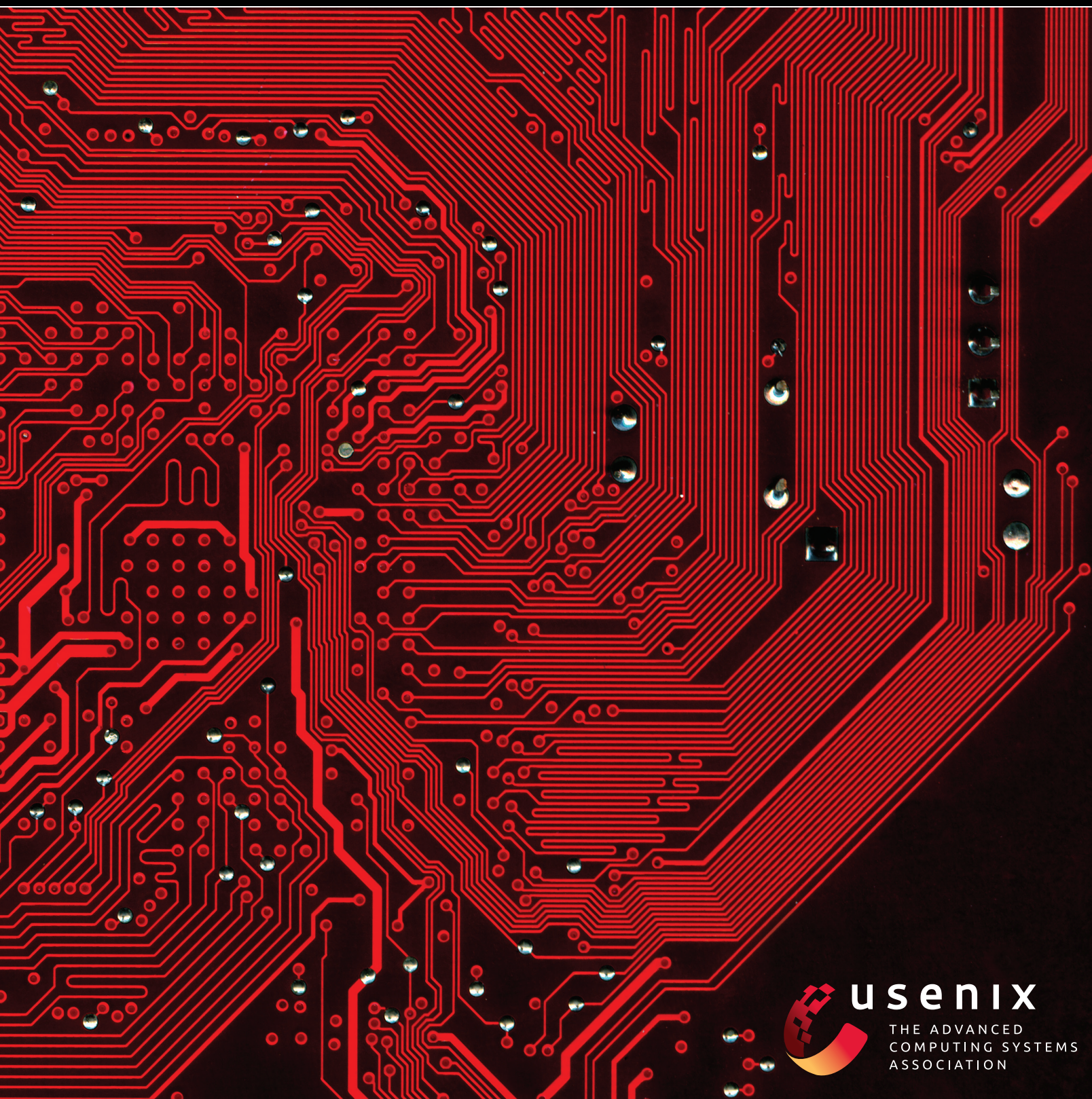
usenix ;login:

JUNE 2012 VOL. 37, NO. 3



usenix
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

usenix ASSOCIATION

UPCOMING EVENTS

21st USENIX Security Symposium (USENIX Security '12)

August 8–10, 2012, Bellevue, WA, USA
http://www.usenix.org/sec12

3rd USENIX Workshop on Health Security and Privacy (HealthSec '12)

CO-LOCATED WITH USENIX SECURITY '12

August 6–7, 2012, Bellevue, WA, USA
http://www.usenix.org/healthsec12

2012 Electronic Voting Technology Workshop/ Workshop on Trustworthy Elections (EVT/ WOTE '12)

CO-LOCATED WITH USENIX SECURITY '12

August 6–7, 2012, Bellevue, WA, USA
http://www.usenix.org/evtwote12

6th USENIX Workshop on Offensive Technologies (WOOT '12)

CO-LOCATED WITH USENIX SECURITY '12

August 6–7, 2012, Bellevue, WA, USA
http://www.usenix.org/evtwote12

5th Workshop on Cyber Security Experimentation and Test (CSET '12)

CO-LOCATED WITH USENIX SECURITY '12

August 6, 2012, Bellevue, WA, USA
http://www.usenix.org/cset12

2nd USENIX Workshop on Free and Open Communications on the Internet (FOCI '12)

CO-LOCATED WITH USENIX SECURITY '12

August 6, 2012, Bellevue, WA, USA
http://www.usenix.org/foci12

7th USENIX Workshop on Hot Topics in Security (HotSec '12)

CO-LOCATED WITH USENIX SECURITY '12

August 7, 2012, Bellevue, WA, USA
http://www.usenix.org/hotsec12

Seventh Workshop on Security Metrics (MetriCon 7.0)

CO-LOCATED WITH USENIX SECURITY '12

August 7, 2012, Bellevue, WA, USA
http://www.usenix.org/metricon70

10th USENIX Symposium on Operating Systems Design and Implementation (OSDI '12)

October 8–10, 2012, Hollywood, CA, USA
http://www.usenix.org/osdi12

Eighth Workshop on Hot Topics in System Dependability (HotDep '12)

CO-LOCATED WITH OSDI '12

October 7, 2012, Hollywood, CA, USA
http://www.usenix.org/hotdep12

2012 Workshop on Power Aware Computing and Systems (HotPower '12)

CO-LOCATED WITH OSDI '12

October 7, 2012, Hollywood, CA, USA
http://www.usenix.org/hotpower12

2012 Workshop on Managing Systems Automatically and Dynamically (MAD '12)

CO-LOCATED WITH OSDI '12

October 7, 2012, Hollywood, CA, USA
http://www.usenix.org/mad12
Submissions due: July 6, 2012

26th Large Installation System Administration Conference (LISA '12)

December 9–14, 2012, San Diego, CA, USA
http://www.usenix.org/lisa12

11th USENIX Conference on File and Storage Technologies (FAST '13)

SPONSORED BY USENIX IN COOPERATION WITH ACM SIGOPS

February 12–14, 2013, San Jose, CA, USA

10th USENIX Conference on Networked Systems Design and Implementation (NSDI '13)

SPONSORED BY USENIX IN COOPERATION WITH ACM SIGCOMM AND ACM SIGOPS

April 3–5, 2013, Lombard, IL, USA

14th Workshop on Hot Topics in Operating Systems (HotOS XIV)

SPONSORED BY USENIX IN COOPERATION WITH THE IEEE TECHNICAL COMMITTEE ON OPERATING SYSTEMS (TCOS)

May 13–15, 2013, Santa Ana Pueblo, NM, USA

FOR A COMPLETE LIST OF ALL USENIX AND USENIX CO-SPONSORED EVENTS,
SEE HTTP://WWW.USENIX.ORG/CONFERENCES

EDITOR
Rik Farrow
rik@usenix.org

MANAGING EDITOR
Jane-Ellen Long
jel@usenix.org

COPY EDITOR
Steve Gilmartin
proofshop@usenix.org

PRODUCTION
Arnold Gatilao
Casey Henderson
Jane-Ellen Long
Michele Nelson

TYPESETTER
Star Type
startype@comcast.net

USENIX ASSOCIATION
2560 Ninth Street, Suite 215,
Berkeley, California 94710
Phone: (510) 528-8649
FAX: (510) 548-5738

http://www.usenix.org
http://www.sage.org

;login: is the official magazine of the USENIX Association. *;login:* (ISSN 1044-6397) is published bi-monthly by the USENIX Association, 2560 Ninth Street, Suite 215, Berkeley, CA 94710.

$90 of each member's annual dues is for a subscription to *;login:*. Subscriptions for nonmembers are $125 per year. Periodicals postage paid at Berkeley, CA, and additional offices.

POSTMASTER: Send address changes to *;login:*, USENIX Association, 2560 Ninth Street, Suite 215, Berkeley, CA 94710.

usenix ;login:

JUNE 2012, VOL. 37, NO. 3

OPINION

Musings

RIK FARROW

Rik is the editor of *;login:*.
rik@usenix.org

In my June 2011 Musings [1], I used the metaphor of an assembly line's parts supply for the hierarchy of storage used with modern processors: cache, memory, disk, and network. I've always been amazed by both assembly lines, and how it is possible that a CPU can get so much work done when it is so much faster than the devices that supply it.

Modern assembly lines are less than a century old, and through the mechanism of YouTube we can easily watch examples of assembly lines at work [2, 3]. In the first example, you can watch a Chevrolet assembly line from 1936, and in the newer one, a BMW line from 2010. One big difference between the two lines is that in the BMW example, the only time you see a person appears to be accidental, just someone walking past the line. In the older line, most people are simply positioning parts, or performing a small set of tasks like several welds or tightening bolts. It is pretty easy to see why owners of a modern line might want to replace people doing boring, repetitive work—even if well-paying—with mindless machines.

Not an Assembly Line

My assembly line metaphor really falls short in a particular way that would have annoyed Alan Turing. A Turing machine mandates having the ability to test results and then branch, something assembly lines do not do. Much work has been done by Intel and other CPU vendors on branch prediction, because CPUs do include miniature assembly lines, called pipelines, which work best when kept filled with both instructions and data. A missed branch invalidates all the work already done by the pipeline, another cause of delay in our speedy processors. These missed branches also mean changes stored in the fastest (L1) cache, meaning more delays waiting for the slower caches and much slower memory.

The inflexibility of real assembly lines is actually a problem for manufacturers. Setting up an assembly line takes time and money, so manufacturers want to continue to use each line for as long as possible. Another problem, similar to the CPU's test and branch, occurs when one stage of the assembly line breaks down: the entire line stops. I got to visit a truck frame assembly line once, while I was visiting a factory that was investigating ways of manufacturing frames without using the traditional assembly line. That company wanted a system that was both more flexible and capable of keeping production going even if one stage breaks down or runs out of supplies.

Locality of instructions is just as important as careful arrangement of data. You are probably aware of several techniques that help with the locality of instructions. Loop unrolling involves doing more work before the (potentially) terminating test instruction. Function inlining replaces a function call with the set of instructions that act on the calling arguments. Both make the code larger, but both reduce the amount of jumping to other locations in memory.

Another technique, which evolved during the late '80s, I believe, was the use of re-entrant libraries. Instead of statically linking libc into every binary, libc gets shared among all programs that use it, through being dynamically linked. This allows just one copy of libc to be present in memory, more likely just the parts of it currently in use, instead of those same parts being loaded into many locations in memory.

Kernel samepage merging [4] is a more recent development that also helps to avoid having multiple copies of the same instructions in memory. Originally developed to reduce the memory footprint of running multiple VMs, where you'd expect there to be lots of duplicate pages if you are running the same OS in many VMs, KSM is now recommended for use even on systems where you are not running VMs.

FlexSC [5] represents another advance in dealing with memory issues caused by system calls. In almost all systems since the dawn of multiprocessing, a special instruction triggers an exception that is handled by the kernel's system call interface. While the kernel executes, it uses its own instructions and data, necessitating the replacement of cache lines used by the currently waiting program. Besides the replacement of user mode instructions and data in various cache levels, other data also gets flushed, such as entries in the data translation look-aside buffer (dTLB), used to convert virtual to physical addresses. In the FlexSC paper, these side effects of a system call exception can decrease the number of instructions completed per CPU clock cycle (IPC) by 20% to 60%.

To fit this into the assembly line metaphor, an exception-based system call is like taking a portion of the parts supply for an assembly line and using it to support a completely different assembly line. This analogy is a forced one, as system calls are actually working on behalf of the program being executed. But it is as if a second assembly line gets called into play, one that shares the same supply stream, and that interference results in less work being completed overall.

The Lineup

We start this issue with an article about a tool for determining the correct balance of memory, disk, and SSD for a server application. Madhyastha et al. explain how their tool, scc, takes into account SLAs and the need for storing and access data, and is able to both reduce cost while suggesting appropriate changes in the proportions of storage devices used. Their implementation of scc is available for download (http://www.cs.ucr.edu/~harsha/scc/).

David Slik describes Cloud Data Management Interface (CDMI), an open protocol for storage data transfer and management for cloud and object storage systems. David first explains the goals of the standard and then demonstrates how its RESTful interface can be used (with curl). CDMI is designed so that storage providers can supply just those portions of the interface that are applicable to the services they provide, and the standard can be extended whenever support for a new interface becomes sufficiently common.

Yanpei Chen and his co-authors revisit work they did for a workshop paper on TCP incast. TCP incast occurs when many servers attempt to reply with data simultaneously, resulting in much lower data transfer. In their article, they explain incast, supply equations for modeling incast, demonstrate the fit of their equations to experimental data, and show how a simple solution can reduce the effects of incast, with several examples of popular distributed systems, including Hadoop.

Robert Escriva and his co-authors, having taken a hard look at current NoSQL solutions, decided that another approach is warranted. They have created HyperDex, a system that provides consistency and reliability guarantees while outperforming popular systems such as Cassandra on benchmarks. In their article, they explain how they use a multi-dimensional space for indexing and node assignment, and how HyperDex manages to be both fast and consistent. Along the way, they highlight issues involving NoSQL solutions.

In my interview with Nathan Milford of Outbrain, I get him to discuss his use of Cassandra. Nathan is both an architect and a sysadmin for his company, and I can tell that he feels comfortable and secure in his decision to use Cassandra, along with several other tools for distributed computing.

Doug Hughes wanted to write about a series of incidents that befell his organization, including the near loss of almost a petabyte of data. Doug describes the diagnostics for several hardware and networking-related problems in terms that will be familiar to most system administrators, and ends each story with some lessons learned. Along the way he describes some useful hardware features.

David Blank-Edelman has decided to discuss the weather in his Perl column. Well, perhaps it would be more accurate to say that he explains how to fetch weather information for particular locations using three different APIs using two different Perl modules for parsing the information. This is not just for weather junkies, but for anyone with the need to pull information out of XML or JSON-encoded data.

David Beazley takes us beyond the basics of Python's lists, sets, and dictionaries, using libraries that will be included with any Python install after versions 2.7 and 3.3. David presents some useful techniques with collections. The Counter and defaultdict objects are dictionaries but with special features, and David provides examples of how to use them, including in analyzing Web logs.

Dave Josephsen begins by being mystified by a coworker who feels that "brothifying" his food will make it more absorbable, but then goes on to tie this concept into making it easier to scale Nagios to more hosts. The Check_MK tool makes collecting multiple checks from a host appear as a single check to Nagios, while simplifying the configuration on the host.

Robert Ferrell was intrigued with the multiple dimensions used in HyperDex and decides to invent his own hyper-dimensional quantum computer. Then he worries about keeping track of data in the cloud, and visualizes techniques for monitoring data as it replicates.

While Elizabeth Zwicky takes a well-deserved break, six other book reviewers tackle six different books. Mark Lamourine discusses *Jenkins: The Definitive Guide,* covering a large book about an ever larger topic, an automated build system. Brandon Ching covers *Webbots, Spiders, and Screen Scrapers,* a second edition about collecting, storing, and processing data collected from the Web, whether from a single page or a wide sweep. Jeff Berg really liked *The Tangled Web,* a book

for anyone who needs to secure Web applications. Evan Teran takes a look at *A Bug Hunter's Diary,* a book targeted at those interested in learning how to find and exploit code vulnerabilities present in various popular software programs. Peter Salus considers *A Culture of Innovation,* a collection of narratives by 19 individuals who have worked at BBN over the years. And, finally, I review *D is for Digital,* certainly the easiest read of this lot, but also a useful book to give to any educated person who wants to know more about computers and networking.

This issue concludes with summaries from the USENIX FAST conference. I've always enjoyed FAST, possibly because it combines hardware and software, academic and commercial research, into a single conference. The scc tool (Madhyastha et al.) was presented as a FAST paper, and there are many other excellent papers summarized in this issue.

Even though the authors of FlexSC have demonstrated that the effects of system calls go well beyond the side-effects of a software interrupt—saving register and other process state, performing the system call (potentially blocking), then restoring process state and continuing to execute in user mode—not much has changed since then. CiteSeerX shows only three citations, and FlexSC has certainly not become a part of the Linux kernel. Yet Apache httpd runs twice as fast with FlexSC, and there are few proposed system-level changes that have such strong, positive effects. I am left wondering whether there are other, better methods for avoiding cache pollution caused by system calls, or are there perhaps architecture advances on the hardware side that will lead to a more efficient system call interface?

Resources

[1] Musings, *;login:*, June 2011, vol. 36, no. 3.

[2] Chevrolet Assembly line in 1936: http://youtu.be/HPpTK2ezxL0.

[3] BMW in 2010: http://youtu.be/KEQdn57Kz1Q.

[4] Kernel samepage merging: http://www.linux-kvm.org/page/KSM.

[5] Livio Soares and Michael Stumm, "FlexSC: Flexible System Call Scheduling with Exception-LessSystem Calls": https://www.usenix.org/conference/osdi10/flexsc-flexible-system-call-scheduling-exception-less-system-calls.

scc

Informed Provisioning of Storage for Cluster Applications

HARSHA V. MADHYASTHA, JOHN C. MCCULLOUGH, GEORGE PORTER, RISHI KAPOOR, STEFAN SAVAGE, ALEX C. SNOEREN, AND AMIN VAHDAT

Harsha V. Madhyastha is an Assistant Professor in Computer Science and Engineering at the University of California, Riverside. His research interests include distributed systems, networking, and security.
harsha@cs.ucr.edu

John C. McCullough is pursuing a PhD in computer science at the University of California, San Diego, and is advised by Professor Alex C. Snoeren. He received an MS in computer science from UCSD (2008) and a BS in computer science from Harvey Mudd College.
jmccullo@cs.ucsd.edu

George Porter is a Research Scientist in the Center for Networked Systems and a member of the Systems and Networking Group at UC San Diego. He received his BS from the University of Texas at Austin and his PhD from the University of California, Berkeley.
gmporter@cs.ucsd.edu

Rishi Kapoor is a PhD student in computer science at the University of California, San Diego. His research interests include cloud computing and computer systems.
rkapoor@cs.ucsd.edu

Stefan Savage is a Professor of Computer Science at the University of California, San Diego. He has a BS in history and reminds his colleagues of this fact anytime the technical issues get too complicated.
savage@cs.ucsd.edu

Alex C. Snoeren is an Associate Professor in the Computer Science and Engineering Department at the University of California, San Diego, where he is a member of the Systems and Networking Research Group. His research interests include operating systems, distributed computing, and mobile and wide-area networking.
snoeren@cs.ucsd.edu

Amin Vahdat is a Principal Engineer at Google and also the SAIC Professor in the Department of Computer Science and Engineering at the University of California, San Diego. Vahdat's research focuses broadly on computer systems, including distributed systems, networks, and operating systems.
vahdat@cs.ucsd.edu

Today, application providers can choose from a range of storage choices to provision their infrastructure for cluster-based applications. Each storage technology presents a different point in a complex tradeoff space of cost, capacity, and performance. To help application providers choose from these alternatives, we developed scc [1] to automate the selection of cluster storage configurations based on a formal specification of applications, hardware, and workloads. Our tool allows administrators to understand how high-level workload characteristics influence the cluster architecture, and in applying scc to several representative deployment scenarios, we show how it can enable 2x–4.5x cost savings when compared to traditional scale-out techniques.

Identifying an appropriate cluster architecture to host a large-scale service is often not straightforward. Given a set of resources to choose from (e.g., as shown in Table 1), an application provider has to answer several questions. What storage technologies should be employed, and how should data be partitioned across them? Where should caching be employed? What types of servers should be chosen to house the selected storage units?

In addition, even if the application's implementation is efficient and there is coarse-grained parallelism in the underlying workload, how will algorithmic shifts in the application or variations in workload affect the appropriate cluster architecture? Our goal is to automate the process of answering these questions, rather than relying solely on human judgment.

Resource	MB/s	IOPS	Watts	Cost
7.2K Disk (500 GB)	90 (R) 90 (W)	125 (R) 125 (W)	5	$213
15K Disk (146 GB)	150 (R) 150 (W)	285 (R) 185 (W)	2.3	$296
SSD (32 GB)	250 (R) 80 (W)	2500 (R) 1000 (W)	2.4	$456
DRAM (1 GB)	12.8K (R) 12.8K (W)	1.6B (R) 1.6B (W)	3.5	$35
CPU core	—	—	20	$137

Server type	Resource Limits	Cost
Server1	4 cores, 1 Gbps network 12GB DRAM, 4 SAS slots	$1400
Server2	16 cores, 10 Gbps network 48GB DRAM, 16 SAS slots	$1850
Server3	32 cores, 10 Gbps network 512GB DRAM, 16 SAS slots	$11000

Table 1: Example set of hardware units input to scc. Cost is price plus energy costs for three years.

In developing scc, we show how to systematically exploit storage diversity, i.e., select among different physical media, local and remote storage, and various caching strategies. First, we determine how the characteristics of applications, workloads, and hardware should be specified in order to automate the selection of cluster configurations. To do so, we study several representative deployment scenarios and identify a parsimonious yet sufficiently expressive set of parameters that capture the tradeoffs offered by different types of storage devices and the varying demands across application components. To characterize applications, we leverage developer knowledge and standard techniques to trace the execution of applications, and, once developed, application models can be reused across deployments. Second, we implement scc, a storage configuration compiler, to take specifications of applications, workloads, and hardware as input, automatically navigate the large space of storage configurations, and zero in on the configuration that meets application SLAs at minimum cost.

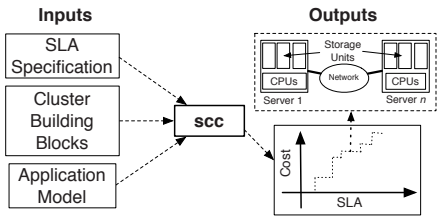


Figure 1: scc takes formal specifications of applications, hardware, and SLA metrics as input. It outputs a cost-versus-SLA distribution, while determining the minimum cost cluster configuration for every SLA value.

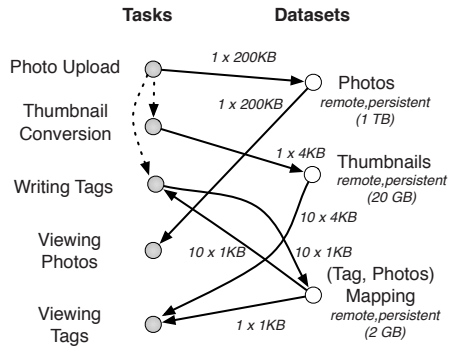


Figure 2: Interaction between tasks and datasets in example photo-sharing application. Edges between tasks and datasets represent I/O with direction differentiating input and output. Dotted edges indicate task dependencies.

Specifying scc's Inputs

As shown in Figure 1, scc takes three inputs: (1) a model of application behavior, specified in part by the application's developer and in part by the administrator deploying the application; (2) characteristics of available hardware building blocks specified by the infrastructure provider; and (3) application performance metrics, i.e., a parameterized service level agreement (SLA) (e.g., a Web service SLA might specify a peak query rate per second). Given these inputs, scc computes how cluster cost varies as a function of the SLA and outputs a low-cost cluster configuration that meets the SLA at each point in the space. scc's output cost vs. SLA value distribution helps administrators decide what performance can be supported cost effectively.

While there has been prior work on similarly configuring storage based on formal specifications of workloads and hardware [2, 3], these prior approaches take as input the workload demands on every component of the application (e.g., the I/O rates to be satisfied by a logical volume of data). In practice, application providers seek to satisfy SLAs that are specified at a higher level. For example, in a photo-sharing Web service, the target may be to cope with a certain rate of photo uploads and downloads. To translate such SLA requirements into demands on individual application components, we need a model of the application.

Our characterization of applications accounts for two aspects: its implementation and the workload in its planned deployment. To capture an application's implementation, we first ask the application's developer to describe its decomposition into compute and storage components, and the interaction between them. We account for various characteristics of these components, such as whether the application runs in multiple phases, the I/O operations it performs in response to particular inputs, and the dependencies between different parts of the application.

For example, Figure 2 depicts the components, and the interaction between them, for an example photo-sharing Web service. Although we place the onus on application developers to formally specify the components of their application, an application's specification is reusable across deployments.

Second, we enable those who deploy an application to annotate the specification of the application's architecture with properties of the expected workload in their deployment. To do so, we require that the compute and I/O characteristics of an application's components, when subjected to the target workload, be determined by running small-scale application benchmarks. We characterize compute components by their memory requirements and storage components by their storage capacity and persistence needs. We also label I/O operations and inter-component dependencies with properties such as the record size being read/written, and whether these operations are synchronous or asynchronous. The former helps differentiate between random and sequential I/O, while the latter determines the application's ability to trade off latency with throughput. Extracting these properties requires tracing the application's execution, now standard practice in resource-intensive performance-critical applications. In the absence of built-in tracing support, systems like Magpie [4] can be leveraged.

Automating the Navigation of the Configuration Space

scc determines the cost versus SLA distribution for a given application deployment by considering the configuration for each point in the distribution independently.

To compute the cluster configuration for a target SLA, scc needs to determine the *architecture* of the cluster (the types of storage media to be used for each dataset and the types of servers used to host storage units and CPUs) and the *scale* at which this architecture must be instantiated (the number of servers, storage units, and CPUs, as well as the level of parallelism of each application task) to meet the SLA.

Guiding Principles

Two key principles help scc identify the right cluster configuration. First, the architecture and scale for every application component can be determined independently when all operations are performed asynchronously, but not when some operations are synchronous. The SLA for any task only specifies the rate at which a task's execution path must run. In the typical case, where a task's execution path contains some operations that block others, scc needs to determine the "division of labor" across these operations that minimizes cost. For example, in a task that reads from an input dataset and then writes to an output dataset, in order to meet the task's SLA it may suffice to provision fast storage for any one of the two datasets; provisioning fast storage for both datasets may unnecessarily result in higher cost due to storage capacity requirements, whereas slow storage for both may incur higher costs in satisfying I/O throughput needs. Hence, scc jointly determines resource requirements across all application components.

Second, since scc provisions for peak load, it prevents over-provisioning by ensuring that at least one resource is bottlenecked on every server at peak load. (If the application provider wants to run the cluster at lower peak utilization, that can be specified as input.) Based on our characterization of hardware, there are four possible bottlenecks on each server: (1) the number of slots, (2) the bandwidth on an I/O controller, (3) the number of CPU cores, (4) network bandwidth.

Algorithm

Driven by the need for joint optimization across components, scc represents each point in the configuration space by the assignment of storage unit types to datasets. This assignment suffices to represent each configuration because, given this information, we can compute the number of storage units of each type and the number of CPUs necessary to meet the SLA. We can then compute the number of servers of each type required to accommodate these resources. As a result, if S is the number of storage choices and D is the number of datasets, scc has to search through a space of $O(S^D)$ configurations; for each dataset, scc can choose any one of the S storage options.

In cases where the configuration space is too large to perform an exhaustive search, scc performs a repeated gradient descent search. We start with a randomly chosen configuration.

In each step, we consider all neighboring configurations—those which differ in exactly one dataset's storage-type assignment—and move to the configuration that still meets the SLA with the maximum decrease in cost. We repeat this step until we find a configuration where all neighbors have higher cost. Since gradient descent can lead to a local minimum, we repeat this procedure multiple times with different randomly chosen initial configurations and settle on the minimum cost output across the multiple attempts. In our evaluation, we have found that repeating the gradient descent 10 times is typically sufficient to find a solution close

to the global minimum. Therefore, even when determining the configuration to satisfy workloads of tens of thousands of queries per second, scc's running time for any particular SLA is within a minute.

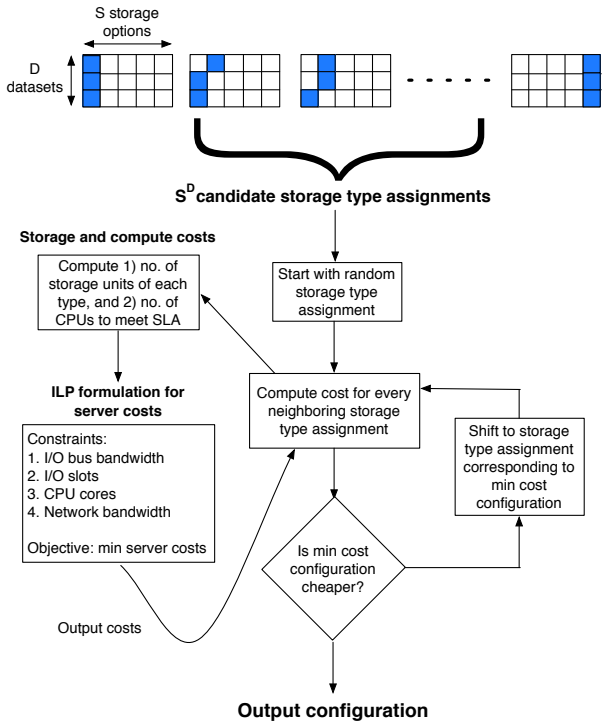


Figure 3: scc represents every configuration by the storage type assignments for each of the application's datasets, and searches through this space with gradient descent (with multiple randomly chosen initial configurations) to find the minimum cost configuration.

At the heart of scc's search of the configuration space (summarized in Figure 3) is a procedure that, given any particular assignment of storage types to datasets, determines a cost-effective set of resources to meet the target SLAs. In this procedure, scc first determines for each remote dataset (i.e., not local to any task) the number of storage units required of the type assigned to the dataset in the configuration state. Second, scc determines the number of CPUs required by every task and the number of storage units of the assigned type needed by the task's local datasets. Finally, it solves a linear integer program to determine the types of servers and number of each kind required to minimize overall cluster cost.

Heterogeneous Configurations Beat Scale-Out

We have applied scc to three distributed applications with distinctly different workload characteristics: (1) a product search Web service modeled on Google Merchant Center, (2) Terasort, a MapReduce job to sort large tuple collections, and (3) a photo-sharing Web service modeled on Flickr. We validated scc by deploying these applications on a range of cluster configurations and measuring application performance on these configurations.

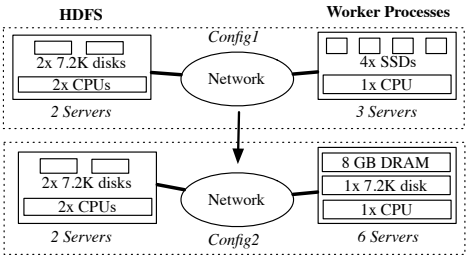
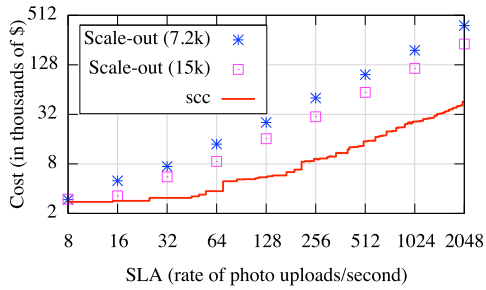


Figure 4: Illustration of transition in minimum cost cluster configuration recommended by scc, when input workload is increased. scc uses heterogeneous architectures to reduce costs in comparison to simply scaling out resources.

In applying scc to these diverse application workloads, we repeatedly find that clusters with heterogeneity—rather than conventional homogeneity—across servers are necessary to optimize cost. The resources required differ across application components due to the varying ratios of capacity, compute, and I/O throughput needs across components. Figure 4 shows an example of how scc's recommended configuration for our example product search Web service changes when the input workload is increased. First, we note that different application components are hosted on servers equipped with different types of storage. Second, the types of hardware resources allocated to the same application component radically change, rather than resources simply being increased in quantity, when the workload is increased.

Transitions in Cost-Optimal Storage Configurations

In applying scc to our exemplar applications, we also find that the most cost-effective cluster architecture depends not only on the application being provisioned but also on the workload and performance requirements. Data that was initially capacity-bound may become I/O-bound at higher loads, calling for shifts from high capacity but slow storage, e.g., disks, to low capacity but fast storage, e.g., SSDs. As a result, cluster configurations output by scc for our exemplar photo-sharing and product search applications result in 2x–4.5x average savings in cost compared to similarly performant scale-out options.



(a)

Uploads/s	Storage unit type		
	Photos	*Thumbnails*	*Tags*
≤ 5	Disk	Disk	Disk
5–25	Disk	Disk	Disk + DRAM
25–330	Disk	SSD	Disk + DRAM
330–930	SSD	Disk + DRAM	Disk + DRAM
930–10k	Disk + DRAM	Disk + DRAM	Disk + DRAM

(b)

Figure 5: (a) Cost versus SLA distribution output by scc for example photo-sharing application, with (b) the corresponding regimes in the cost-effective architecture. Simply scaling out alternate configurations inflates cost by 3x–4.5x on average.

As an example, Figure 5(a) shows the cost distribution output by scc across a range of SLA values for our photo-sharing application. Perhaps surprisingly, no huge spikes are observed in this distribution; this is because scc balances costs across the kind of storage, the number of CPUs, and the number of machines provisioned. Rather than adding more machines of the same type, the cluster architecture transitions to faster storage as the SLA becomes more stringent, with transitions in storage type for different datasets seen at different SLA values.

Figure 5(b) highlights these transitions. Note that the quantity in which different types of resources are provisioned varies within each architecture regime specified by every row in the table.

We further compare the cost output by scc with the cost associated with a scale-out approach. We compare the scc configuration to the cases where the building block is based around: (1) storage servers with four 7.2k-RPM disks (the cost-optimal storage type for all datasets at the lowest SLA), and (2) servers with four 15k-RPM disks. In either case, more storage servers are added as the required rates increase. Figure 5(a) shows that the costs in both cases are significantly greater than with scc, incurring between 3 and 4.5 times more cost (note the logarithmic y-axis). Thus, simply scaling out a homogeneous configuration that is cost-effective at low loads can result in significant cost inflation at higher loads.

How Robust Are scc's Recommendations?

scc's output cluster configuration for a target SLA is a function of both the SLA and the values specified for the various attributes in the application and hardware specifications. In practice, an administrator may not have precise values for all attributes due to incomplete knowledge of the application workload, uncertainty of hardware costs, or measurement inaccuracy in benchmarking the application.

Attribute	**Range with same architecture**		
	Lowest value	Input value	Highest value
Avg. photo size	50 KB	200 KB	850 KB
Avg. thumbnail size	1 KB	4 KB	30 KB
SSD unit price	$200	$450	$900

Dataset	**Most sensitive to what change in hardware costs?**
Photos	20% drop in $ of 7.2K-RPM disk
Thumbnails	92% drop in $ of DRAM
Tags	31% drop in $ of 15K-RPM disk

Table 2: (a) Robustness of scc's output with respect to input values for a sample set of attributes; (b) the change in hardware costs to which scc's storage decision for each dataset is most sensitive.

scc is naturally built to cope with such uncertainty. For every attribute in the input specifications, scc varies the value of the attribute in the neighborhood of the initially specified value. For each attribute, it then outputs the range of values for that attribute wherein the cost-effective cluster architecture, i.e., the types of resources

assigned to different application components, remains unchanged; variance of the attribute's value within this range can be handled by simply adding more resources of the same type. Outside of that range, the cluster will need to be revamped with a different type of resource for some application component, a more onerous undertaking. For example, we consider our example photo-sharing service with an SLA of 100 uploads/s, 300 photo views/s, and 100 tag views/s. Table 2(a) shows the value ranges output by scc for a few attributes, within which the cluster architecture is robust to change. For example, we see that as long as the average photo size remains between 50 KB and 850 KB, the cluster architecture remains the same as that obtained with the input value of 200 KB.

Furthermore, scc can also evaluate the sensitivity of its choice of storage configuration for every dataset in the application. For example, consider our photo-sharing Web service again with the same input SLA as above. Based on current hardware costs, scc determines that photos be stored on 15k-RPM disks, thumbnails be stored on SSDs, and tags be stored persistently on 7.2k-RPM disks and cached in DRAM, in order to meet the SLA at minimum cost. However, these recommendations are likely to change as prices for storage units drop. scc can determine the robustness of its storage option choice in response to such changes in hardware prices. To do so, it varies the price of every type of storage unit from its input value down to 0, and notes the inflection points at which the optimal storage choice for some dataset changes. Based on this analysis, it can determine, for every dataset, that change in hardware price to which the current storage choice for the dataset is most sensitive. Table 2(b) shows that while the storage choices for photos and tags are sensitive to relatively small reductions in the prices for 7.2k-RPM and 15k-RPM disks, scc's recommendation of storing thumbnails on SSDs is very robust to price fluctuations.

Conclusion

The primary thesis of our work is that the choice of cluster hardware for an application should be informed by the interaction between the application's behavior and the properties of hardware. Rather than relying on human judgment to do so, we developed scc to compile formal specifications of these inputs into cost-effective cluster configurations. We have applied scc to a range of application workloads and storage options to demonstrate that scc captures sufficient detail to identify the appropriate hardware at any given scale. We find that scc often recommends heterogeneous cluster architectures that result in significant cost savings compared to traditional scale-out approaches.

Our implementation of scc is available for download at http://www.cs.ucr.edu/~harsha/scc/.

Acknowledgments

This work was supported in part by a NetApp Faculty Fellowship and through a grant from the National Science Foundation (CNS-1116079).

References

[1] H.V. Madhyastha, J.C. McCullough, G.M. Porter, R. Kapoor, S. Savage, A.C. Snoeren, and A. Vahdat, "scc: Cluster Storage Provisioning Informed by Application Characteristics and SLAs," in FAST, 2012.

[2] G.A. Alvarez, E. Borowsky, S. Go, T.H. Romer, R.A. Becker-Szendy, R.A. Golding, A. Merchant, M. Spasojevic, A.C. Veitch, and J. Wilkes, "Minerva: An Automated Resource Provisioning Tool for Large-Scale Storage Systems," ACM Transactions on Computer Systems, 2001.

[3] J. Wilkes, "Traveling to Rome: QoS Specifications for Automated Storage System Management," in IWQoS, 2001.

[4] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier, "Using Magpie for Request Extraction and Workload Modeling," in OSDI, 2004.

An Introduction to CDMI

DAVID SLIK

David Slik is the Technical Director for Object Storage at NetApp and is a co-chair of the SNIA Cloud Storage Technical Working Group. He participated in the creation of the XAM and CDMI standards and has architected and developed large-scale distributed object storage systems deployed worldwide as a key component of mission-critical enterprise applications.
dslik@netapp.com

In early 2009, recognizing the growing importance of cloud storage, and building on top of the earlier XAM object storage standard [1], the Storage Networking Industry Association (SNIA) [2] started a new technical working group with the primary goal of creating an industry standard for cloud storage. The result was the Cloud Data Management Interface (CDMI) [3], which was released as a formal industry standard in 2011 and is currently in the process of becoming an ISO/IEC standard.

The creation of the CDMI standard has been a collaborative effort, with contributions from over a hundred storage vendors, end users, and university researchers. All of the major enterprise storage vendors contributed to the standard, including Dell, Cisco, EMC, HP, Hitachi Data Systems, Huawei, IBM, Intel, LSI, NetApp, Oracle, Symantec, and VMware [4]. This represents a unique cross-industry endorsement of cloud storage, and the results are clearly visible in the breadth of use cases that CDMI is able to address.

In the year following the initial publication of CDMI, the SNIA has published an errata release of the standard and held four plugfests to demonstrate interoperability and conformance of open source, research, and commercial implementations. In the coming year, additional milestones will be reached, with major storage vendors bringing CDMI-compatible systems to market, and work ongoing to add CDMI to open source platforms, including OpenStack [5].

Design Principles

The following principles guided the design of the CDMI protocol:

- **Complementary**—A key design principle is that CDMI is designed to complement, not replace, existing NAS, SAN, and object protocols. Traditional file systems and LUNs can be managed out-of-band using CDMI, in conjunction with access via NAS and SAN protocols. CDMI can also be used as a self-service management and/or access protocol alongside other object protocols, such as OpenStack's Swift. CDMI adopts many best-practice designs from existing protocols, such as NFSv4 ACLs, XAM globally unique identifiers, RESTful HTTP, and JSON structured metadata. Furthermore, as a protocol specification, CDMI places minimal restrictions on how servers are implemented, allowing it to be easily added to existing file, object, and cloud servers.
- **Simple**—In order to foster adoption and reduce the cost required to implement CDMI, the protocol is designed to be as simple as possible. By building on top

of HTTP, standard libraries and language constructs can be used, reducing the need for cloud libraries and allowing direct access by JavaScript browser-based clients. Using standard HTTP authentication and security mechanisms avoids complex header calculations. And providing the ability to start simple and add complexity only when needed reduces the learning curve and simplifies client code. Storing and retrieving your first CDMI object is as easy as:

```
demo$ curl -X PUT -d 'Hello CDMI World' -k http://127.0.0.1:18080/hello.txt
demo$ curl -X GET -v -k http://127.0.0.1:18080/hello.txt
```

This simplicity makes CDMI very script-friendly, allowing it to be easily used to create structured data repositories. At a recent coding challenge, a distributed CDMI-based temperature monitoring and reporting system was developed in hours, complete with a Web-based JavaScript front-end that retrieved data directly from the repository.

- **Extensible**—Recognizing that cloud storage is still in its infancy and that custom features are often required and desired, CDMI was designed from the ground up to allow functionality to be added to the standard without breaking client interoperability. CDMI allows clients to dynamically discover what features a server implements, and it allows clients to discover profiles of capabilities required to perform common use cases. The SNIA also has defined a process by which emerging extensions can be documented, and once multi-vendor implementations have been demonstrated, they can then be incorporated into the next version of the standard.

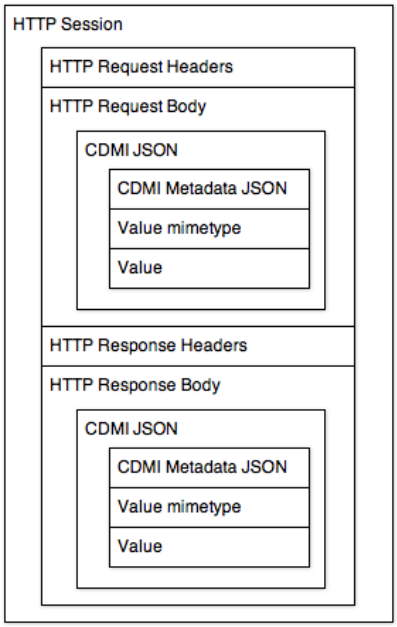


Figure 1: CDMI requests and responses are embedded in HTTP sessions.

CDMI as a Storage Protocol

CDMI is an encapsulation protocol based around RESTful HTTP. Representational State Transfer, or REST, was initially described by Roy Fielding in Chapter five of his PhD dissertation [6], and codifies a series of architectural patterns for the creation of Web-scale distributed systems. The key principles of RESTful architectures include stateless communication, idempotent operations with minimal side effects resulting from repeating a given transaction, and the use of negotiated "representations" for entities that are transferred between clients and servers.

CDMI defines five basic representations (content-types), described in RFC 6208, which are transferred between a client and a server in HTTP Request Bodies and HTTP Response Bodies, as illustrated in Figure 1.

While CDMI 1.0 defines JSON-based representations, the standard is structured such that XML representations can easily be added.

CDMI also defines "Non-CDMI" interactions, where the value is directly transferred in the HTTP request and response body. This provides the ability for a CDMI server to act as a standard Web server to unmodified Web clients and browsers.

A Non-CDMI Request for a stored data object returns a standard HTTP response:

```
demo$ curl -X GET -v -k http://127.0.0.1:18080/hello.txt
* Connected to 127.0.0.1 (127.0.0.1) port 18080 (#0)
> GET /hello.txt HTTP/1.1
> User-Agent: curl/7.19.7 (universal-apple-darwin10.0) libcurl/7.19.7
OpenSSL/0.9.8r zlib/1.2.3
```

```
> Host: 127.0.0.1:18080
> Accept: */*
>
< HTTP/1.1 200 OK
< Content-Type: text/plain
< Content-Length: 16
<
Hello CDMI World
* Closing connection #0
```

A CDMI Request for the same stored data object returns the CDMI JSON representation, which includes additional information about the stored object:

```
demo$ curl -X GET -v --header 'Accept: application/cdmi-object'--header
'X-CDMI-Specification-Version: 1.0.1' -k http://127.0.0.1:18080/hello.txt
* Connected to 127.0.0.1 (127.0.0.1) port 18080 (#0)
> GET /hello.txt HTTP/1.1
> User-Agent: curl/7.19.7 (universal-apple-darwin10.0) libcurl/7.19.7
OpenSSL/0.9.8r zlib/1.2.3
> Host: 127.0.0.1:18080
> Accept: */*
> Content-Type: application/cdmi-object
> X-CDMI-Specification-Version: 1.0.1
>
< HTTP/1.1 200 OK
< Content-Type: application/cdmi-object
< Content-Length: 1033
< Connection: close
< X-CDMI-Specification-Version: 1.0.1
<
{
  "objectType": "application/cdmi-object",
  "objectID": "00007ED90012E02F466C7574746572736879",
  "objectName": "hello.txt",
  "parentURI": "/",
  "parentID": "00007ED9001OC2A44D4C503A46694D21",
  "domainURI": "/cdmi_domains/",
  "capabilitiesURI": "/cdmi_capabilities/dataobject/",
  "completionStatus": "Complete",
  "mimetype": "text/plain",
  "metadata": {
    "cdmi_ctime": "2012-03-20T18:53:46.238543",
    "cdmi_mtime": "2012-03-20T18:53:46.238543",
    "cdmi_mcount": "1",
    "cdmi_owner": "root",
    "cdmi_group": "root",
    "cdmi_acl": [
      {
        "identifier": "OWNER@",
        "acetype": "ALLOW",
        "aceflags": "OBJECT_INHERIT, CONTAINER_INHERIT, INHERITED",
        "acemask": "ALL_PERMS"
```

```
    },
    {
      "identifier": "AUTHENTICATED@",
      "acetype": "ALLOW",
      "aceflags": "OBJECT_INHERIT, CONTAINER_INHERIT, INHERITED",
      "acemask": "READ"
    }
   ],
   "cdmi_size": "16"
  },
  "valuerange": "0-15",
  "valuetransferencoding": "utf-8",
  "value": "Hello CDMI World"
}
* Closing connection #0
```

In the above data object retrieval example, the meaning of the JSON fields in the HTTP response body is listed in Table 1.

JSON Field	**Description**
objectType	Indicates the type of object described in the JSON body. CDMI mimetypes are defined in RFC 6208.
objectID	Every CDMI object has a globally unique identifier that remains constant over the life of the object.
objectName	The name of the object. Present only if the object is stored in a container.
parentURI	The URI of the container where the object is stored. Present only if the object is stored in a container.
parentID	The object ID of the parent container when stored in a container.
domainURI	The URI of a domain object corresponding to the administrative domain that owns the object.
capabilitiesURI	The URI to a capabilities object describing what can be done to the object.
completionStatus	Indicates whether the object is complete or is in the process of being created. This is used for long-running operations.
mimetype	Indicates the mimetype of the value of the data object.
metadata	System and user-provided metadata, in JSON format. Examples of metadata include system properties such as creation time, size, owner, ACLs. Additional user-specified metadata is also stored.

valuerange	Indicates the byte range returned in the value field.
valuetransferencoding	Indicates the encoding of the value field. CDMI supports both UTF-8 and base64 encodings.
value	The data stored in the object.

Table 1: JSON fields returned in an example CDMI Data Object retrieval

Each CDMI object type defines different JSON fields that, in turn, define how objects are set and retrieved, with data objects defined in clause 8, containers defined in clause 9, domains defined in clause 10, queues defined in clause 11, and capabilities defined in clause 12.

Example Client Use Cases

To provide a real-world example, let us suppose that we have been tasked with creating a distributed temperature monitoring system. Our requirements are to sample the temperature of the processor of each of our servers, storing second granular samples every minute to a repository, and providing a Web-based front-end allowing users to visualize temperature across the datacenter.

Using CDMI, a small daemon would be written that runs on each server. This daemon collects 60 samples of data aligned to a minute, and stores it as a CDMI object, including user metadata for the start time, end time, server name, system load average, and processor type.

A JavaScript-based Web page is also served from the CDMI server. When accessed, the JavaScript program is run within the browser and performs a CDMI query based on the user metadata stored in the objects. For example, if a user wishes to see temperature for a given time range, the metadata is used to return only the temperature values within those time ranges. Various visualizations are then generated based on the temperature values returned in the query results.

A second example is a scalable cloud-based OCR system. Multiple scanning workstations scan documents and store them as a data objects. Once scanned, the object ID for each data object is enqueued into a CDMI queue object.

Multiple OCR engines are then instantiated within one or more compute clouds, with the number of instances dynamically varying based on the current size of the CDMI queue. Each OCR engine checks out a scan from the queue, performs OCR processing, and generates a new data object containing a PDF. Based on notifications of the creation of these PDFs, email notifications are then sent to the originator of the scan, or the PDF shows up in the user's home directory.

CDMI Functionality

The following sections provide a survey of the functionality defined by the CDMI standard. To learn more, additional details and examples can be found in the CDMI standard document [3].

Client-to-Cloud Data Transfer

The first area of scope for the CDMI standard is providing standardized methods for clients to exchange data for storage in clouds.

CAPABILITIES DISCOVERY

The CDMI standard mandates that every CDMI server shall provide the ability for clients to discover what optional parts of the standard are implemented in a given server. As the CDMI standard addresses many different cloud-related use cases, allowing an implementer to select the subset of CDMI's functionality specific to their target applications avoids imposing additional development costs for unneeded functionality. For example, a read-only cloud service is free to only implement functionality related to retrieval of stored data, whereas a cloud using CDMI to manage block storage LUNs would only need to implement containers and the ability to define exports.

Clients discover which parts of the CDMI standard are implemented by inspecting published "capabilities." Profiles are also defined to allow clients to determine if logical sets of related capabilities are implemented.

DATA OBJECTS

CDMI data objects are similar to files, and store a value along with metadata. Data objects can be accessed by ID and/or name and support partial retrievals and updates.

CONTAINERS

CDMI container objects are similar to directories and contain named children that can be listed, along with metadata. Containers can be accessed by ID and/or name and support partial listing of children. Traditional hierarchies can be created using sub-containers.

QUEUES

CDMI queue objects are similar to data objects, where multiple values can be stored in a first-in/first-out manner. Queues are typically used to provide persistent inter-process communication structures between distributed programs running in the cloud, and are also used as a foundation for advanced CDMI functionality such as query and notification.

NOTIFICATION

CDMI allows clients to request that when stored objects are created, retrieved, updated, or deleted, notifications of these changes are enqueued into a client-created CDMI queue. Clients can specify the characteristics of the objects for which notifications are generated, based on metadata matching criteria, and can specify which events are of interest and what information is to be returned in each generated notification. Notifications allow powerful workflows to be created, where loosely coupled programs can react to events in the cloud, such as performing transcoding, format conversion, sending notifications, and synchronizing between multiple storage systems.

QUERY

CDMI allows clients to perform a query to find all stored objects that match a set of client-specified metadata matching criteria. Clients can specify which objects are included in the query results and what information from each object is to be included for each query result found. Query allows clients to quickly locate stored

objects, which can be used for further processing or displayed as results to end users.

ACCESS CONTROL

Access to CDMI objects is controlled by ACLs, which define the visibility, read, write, and deletion privileges. The mapping of client credentials to the ACL principal is managed via CDMI domains, which allows content administered by different organizations to co-exist within a single namespace.

Client-to-Cloud Management

The second area of scope for the CDMI standard is providing standardized methods for clients to manage data stored in clouds.

ADMINISTRATIVE DOMAINS

CDMI introduces the concept of Cloud Domains, which permit clients to manage credential to identity mapping (think nsswitch for the cloud) and provide accounting and summary usage information. Domains are hierarchical, which permits tenant and subtenant models, along with delegated administration. Every stored object belongs to a single domain, which controls how the object is accessed and determines who has administrative control over the object.

DATA SYSTEM METADATA

In order to provide a channel that enables clients to express the data services they desire for content stored in the cloud and to give cloud storage system feedback to a client indicating which services are being offered, CDMI introduces a specialized type of metadata known as Data System Metadata (DSM). Instead of providing low-level details about storage, such as RAID level, DSM is expressed in terms of service level objectives, such as desired throughput, latency, and protection.

A client specifies the desired DSM characteristics on individual objects or on containers of objects, which then propagate their DSM to all child containers and data objects. This provides hints about how data should be stored internally within the cloud, allowing a cloud to optimize its internal operations and to charge based on services requested and thus delivered.

The cloud can then create corresponding DSM feedback items, known as "_provided" metadata items, that indicate to a client which actual service the client is receiving. For example, if a client requests three-way replication by setting the "cdmi_data_redundancy" DSM to the value "3", but the system can only provide two-way replication, the "cdmi_data_redundancy_provided" DSM would have the value "2".

A complete list of standardized DSM items can be found in clause 16.4 of the CDMI standard.

RETENTION

CDMI defines a series of DSM that allow restrictions to be placed on stored objects for compliance, eDiscovery, and regulatory purposes. Objects can be placed under retention, meaning they cannot be altered or deleted; can be placed under legal hold

(preventing deletion or modification); and can be automatically deleted when they are no longer under retention periods or any holds.

SNAPSHOTS

CDMI allows clients to trigger the creation of snapshots on a container-based granularity. Snapshots can be accessed through the CDMI interface, and provide read-only access.

EXPORTS

DMI defines the ability to export CDMI containers via standard NAS or SAN protocols. The same approach can be extended to export CDMI namespaces via other cloud protocols, or export queues via other queuing protocols such as AMQP [7]. When combined with cloud computing standards such as OCCI [8] and CIMI [9], CDMI can provide full storage management services for both traditional block and file services accessed by cloud computing resources.

LOGGING

CDMI defines a standardized queue-based mechanism by which clients can receive cloud logging and audit data. This is especially important when a cloud acts as a proxy or broker and logging data must be aggregated or translated. The CDMI standard does not define the contents of log messages originating from clouds.

Cloud-to-Cloud Interactions

The third and final area of scope for the CDMI standard is providing standardized methods for clouds to transfer data with other clouds, both as a result of client requests and automatically.

GLOBALLY UNIQUE IDENTIFIERS

Every CDMI object has a globally unique identifier that remains constant for the life of the object, even as objects are moved or replicated across systems provided by different vendors. This enables location-independent access and allows content to be migrated and replicated without requiring updates to the client's knowledge about how to access the stored data, as the identifier remains constant.

SERIALIZATION/DESERIALIZATION

CDMI objects can be serialized into a JSON format that can be used to transport objects and their metadata between systems. This provides a portable representation for backup and restore, as well as cloud-to-cloud transfer, even if it entails shipping hard drives or tapes storing the data.

CLOUD-TO-CLOUD DATA MOVEMENT

CDMI defines primitives that allow a client to request that a new object be created from an existing object in the same or a different cloud. This allows the destination cloud to retrieve the object directly from the source cloud (using credentials from the CDMI domain, or distributed authentication systems such as OAuth), avoiding the need to transfer the data to and from the client. This also enables clouds to provide transparent proxy and broker functions, while still allowing client access to the underlying clouds themselves.

AUTHENTICATION DELEGATION

CDMI allows resolution of user credentials and mapping to ACL principals to be delegated, allowing CDMI systems to be easily interfaced both with local identity management systems such as AD and LDAP and with emerging federated identity systems.

Future Directions

The SNIA Cloud Technical Working Group encourages interested parties to join the group, participate in plugfests, and submit extensions to the CDMI protocol. Following review by the technical working group, extensions are published for public review. Once two interoperable implementations of an extension are demonstrated at a plugfest, the extension can then be voted on for incorporation into the next version of the CDMI standard.

CDMI extensions currently under public review can be found at the SNIA Web site [10].

References

[1] XAM standard: http://snia.org/forums/xam/technology/specs.

[2] Storage Networking Industry Association: http://www.snia.org/.

[3] CDMI standard: http://snia.org/sites/default/files/CDMI_SNIA_Architecture_v1.0.1.pdf.

[4] Cloud Storage Initiative member companies: http://www.snia.org/forums/csi.

[5] CDMI submission to Swift: https://blueprints.launchpad.net/swift/+spec/cdmi.

[6] REST architecture style: http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm.

[7] Advanced Message Queuing Protocol: http://www.amqp.org/.

[8] OCCI standard: http://occi-wg.org/about/specification/.

[9] CIMI standard: http://dmtf.org/sites/default/files/standards/documents/DSP0263_1.0.0c.pdf.

[10] Draft CDMI extensions: http://snia.org/tech_activities/publicreview/cdmi.

Understanding TCP Incast and Its Implications for Big Data Workloads

YANPEI CHEN, REAN GRIFFITH, DAVID ZATS, ANTHONY D. JOSEPH, AND RANDY KATZ

Yanpei Chen is a fifth-year PhD student at the University of California, Berkeley. His research focuses on workload-driven design and evaluation of large-scale Internet datacenter systems, and includes industrial collaborations with Cloudera, NetApp, and Facebook. He is a member of the Algorithms, Machines, and People's Laboratory, and he holds a National Science Foundation Graduate Research Fellowship.
ychen2@eecs.berkeley.edu

Rean Griffith is a Staff Engineer in the CTO's office at VMware. Prior to joining VMware in 2010, he was a post-doc in the RAD Lab at UC Berkeley. He received his PhD in computer science from Columbia University in 2008. His research interests include distributed systems, operating systems, adaptive systems and networks, control systems, performance and reliability modeling, and the application of statistical machine learning to resource management problems.
rean@vmware.com

David Zats received his BS in computer science and engineering from UCLA in 2007 and his MS in electrical engineering and computer sciences from UC Berkeley in 2009. He is currently working toward his PhD at UC Berkeley, where his research focus is on reducing performance variability in datacenter networks.
dzats@eecs.berkeley.edu

Anthony D. Joseph is a Chancellor's Associate Professor in Electrical Engineering and Computer Science at UC Berkeley. He is developing adaptive techniques for cloud computing, network and computer security, and security defenses for machine learning–based decision systems. He also co-leads the DETERlab testbed, a secure scalable testbed for conducting cybersecurity research.
adj@eecs.berkeley.edu

Randy H. Katz is the United Microelectronics Corporation Distinguished Professor in Electrical Engineering and Computer Science at UC Berkeley, where he has been on the faculty since 1983. His current interests are the architecture and design of modern Internet datacenters and related large-scale services.
randy@cs.Berkeley.edu

TCP incast is a recently identified network transport pathology that affects many-to-one communication patterns in datacenters. It is caused by a complex interplay between datacenter applications, the underlying switches, network topology, and TCP, which was originally designed for wide area networks. Incast increases the queuing delay of flows, and decreases application level throughput to far below the link bandwidth. The problem especially affects computing paradigms in which distributed processing cannot progress until all parallel threads in a stage complete. Examples of such paradigms include distributed file systems, Web search, advertisement selection, and other applications with partition or aggregation semantics [25, 18, 5].

There have been many proposed solutions for incast. Representative approaches include modifying TCP parameters [27, 18] or its congestion control algorithm [28], optimizing application level data transfer patterns [25, 21], switch level modifications such as larger buffers [25] or explicit congestion notification (ECN) capabilities [5], and link layer mechanisms such as Ethernet congestion control [3, 6]. Application level solutions are the least intrusive to deploy, but require modifying each and every datacenter application. Switch and link level solutions require modifying the underlying datacenter infrastructure and are likely to be logistically feasible only during hardware upgrades.

Unfortunately, despite these solutions, we still have no quantitatively accurate and empirically validated model to predict incast behavior. Similarly, despite many studies demonstrating incast for micro-benchmarks, we still do not understand how incast impacts application level performance subject to real life complexities in configuration, scheduling, data size, and other environmental and work-

load properties. These concerns create justified skepticism on whether we truly understand incast at all, whether it is even an important problem for a wide class of workloads, and whether it is worth the effort to deploy various incast solutions in front-line, business-critical datacenters.

We seek to understand how incast impacts the emerging class of big data workloads. Canonical big data workloads help solve needle-in-a-haystack type problems and extract actionable insights from large-scale, potentially complex and unformatted data. We do not propose in this article yet another solution for incast. Rather, we focus on developing a deep understanding of one existing solution: reducing the minimum length of TCP retransmission time out (RTO) from 200 ms to 1 ms [27, 18]. We believe TCP incast is fundamentally a transport layer problem; thus, a solution at this level is best.

The first half of this article develops and validates a quantitative model that accurately predicts the onset of incast and TCP behavior both before and after. The second half of this article investigates how incast affects the Apache Hadoop implementation of MapReduce, an important example of a big data application. We close the article by reflecting on some technology and data analysis trends surrounding big data, speculate on how these trends interact with incast, and make recommendations for datacenter operators.

Toward an Analytical Model

We use a simple network topology and workload to develop an analytical model for incast, shown in Figure 1. This is the same setup as that used in prior work [25, 27, 18]. We choose this topology and workload to make the analysis tractable.

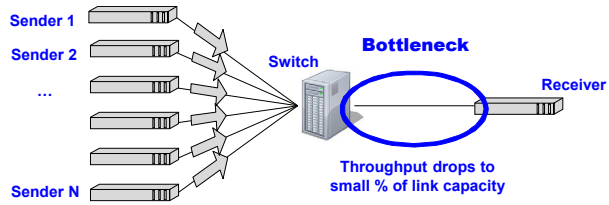


Figure 1: Simple setup to observe incast

The workload is as follows. The receiver requests k blocks of data from a set of *N* storage servers—in our experiments k = 100 and *N* varies from 1 to 48. Each block is striped across *N* storage servers. For each block request received, a server responds with a fixed amount of data. Clients do not request block k+1 until all the fragments of block k have been received. This leads to a *synchronized read pattern* of data requests. We reuse the storage server and client code in [25, 27, 18]. The performance metric for these experiments is *application-level goodput*, i.e., the total bytes received from all senders divided by the finishing time of the *last* sender.

We conduct our experiments on the DETER Lab testbed [12], where we have full control over the non-virtualized node OS, as well as the network topology and speed. We used 3 GHz dual-core Intel Xeon machines with 1 Gbps network links. The nodes run standard Linux 2.6.28.1. This was the most recent mainline Linux distribution in late 2009, when we obtained our prior results [18]. We present results using both a relatively shallow-buffered Nortel 5500 switch (4 KB per port) and a more deeply buffered HP Procurve 5412 switch (64 KB per port).

Flow Rate Models

The simplest model for incast is based on two competing behaviors as we increase *N*, the number of concurrent senders. The first behavior occurs before the onset of incast and reflects the intuition that goodput is the block size divided by the transfer time. Ideal transfer time is just the sum of a round trip time (RTT) and the ideal send time. Equation 1 captures this idea.

$$Goodput_{beforeIncast} = idealGoodputPerSender \times N$$

$$= \frac{blockSize}{idealTransferTime} \times N$$

$$= \frac{blockSize}{RTT + \frac{blockSize}{perSenderBandwidth}} \times N$$

$$= \frac{blockSize}{RTT + \frac{blockSize \times N}{linkBandwidth}} \times N \qquad (1)$$

Incast occurs when there are some $N > 1$ concurrent senders, and the goodput drops significantly. After the onset of incast, TCP retransmission time out (RTO) represents the dominant effect. Transfer time becomes RTT + RTO + ideal send time, as captured in Equation 2. The goodput collapse represents a transition between the two behavior modes.

$$Goodput_{incast} = goodputPerSender \times N$$

$$= \frac{blockSize}{RTO + idealTransferTime} \times N$$

$$= \frac{blockSize}{RTO + RTT + \frac{blockSize}{perSenderBandwidth}} \times N$$

$$= \frac{blockSize}{RTO + RTT + \frac{blockSize \times N}{linkBandwidth}} \times N \qquad (2)$$

Figure 2 gives some intuition with regard to Equations 1 and 2. We substitute blockSize = 64KB, 256 KB, 1024 KB, and 64 MB, as well as RTT = 1 ms, and RTO = 200 ms. Before the onset of incast (Equation 1), the goodput increases as *N* increases, although with diminishing rate, asymptotically approaching the full link bandwidth. The curves move vertically upwards as block size increases. This reflects the fact that larger blocks result in a larger fraction of the ideal transfer time spent transmitting data, versus waiting for an RTT to acknowledge that the transmission completed. After incast occurs (Equation 2), RTO dominates the transfer time for small block sizes. Again, larger blocks lead to RTO forming a smaller ratio versus ideal transmission time. The curves move vertically upwards as block size increases.

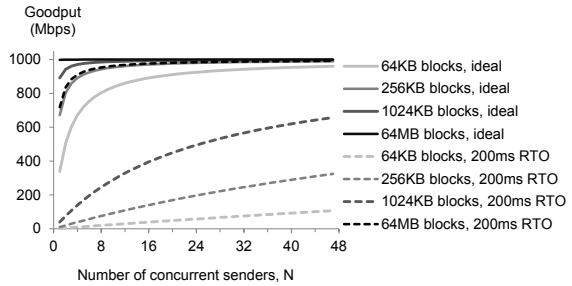


Figure 2: Flow rate model for incast, showing ideal behavior (solid lines, Equation 1) and incast behavior caused by RTOs (dotted lines, Equation 2). The incast goodput collapse comes from the transition between the two TCP operating modes.

Empirical Verification

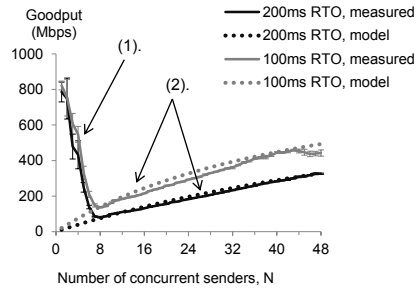


Figure 3: Empirical verification of flow rate incast model. Error bars represent 95% confidence interval around the average of five repeated measurements. This shows that (1) incast goodput collapse begins at N = 2 senders, and (2) behavior after goodput collapse verifies Equation 2.

This model matches well with our empirical measurements. Figure 3 superpositions the model on our previously presented data in [18]. There, we fix block size at 256 KB and set RTO to 100 ms and 200 ms. The switch is a Nortel 5500 (4 KB per port). For simplicity, we use RTT = 1 ms for the model. Goodput collapse begins at $N = 2$, and we observe behavior for Equation 2 only. The empirical measurements (solid lines) match the model (dotted-lines) almost exactly.

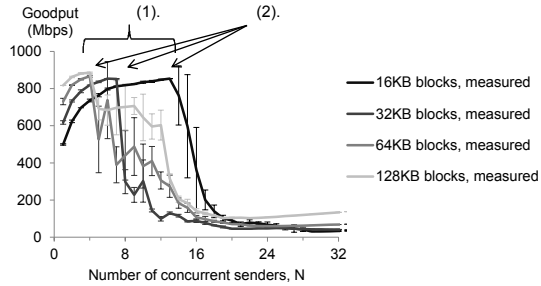


Figure 4: Empirical verification of flow rate TCP model before onset of incast. RTO is 200 ms. Error bars represent 95% confidence interval around the average of five repeated measurements. This shows (1) that behavior before goodput collapse verifies Equation 1, and (2) the onset of incast goodput collapse predicted by switch buffer overflow during slow start (Equation 3).

We use a more deeply buffered switch to verify Equation 1. As we discuss later, the switch buffer size determines the onset of incast. Figure 4 shows the behavior using the HP Procurve 5412 switch (64 KB per port). Behavior before goodput collapse qualitatively verifies Equation 1—the goodput increases as N increases, although with diminishing rate; the curves move vertically upwards as block size increases. We can see this graphically by comparing the curves in Figure 4 before the goodput collapse to the corresponding curves in Figure 2.

Takeaway: Flow rate model captures behavior before onset of incast. TCP RTO dominates behavior after onset of incast.

Predicting the Onset of Incast

Figure 4 also shows that goodput collapse occurs at different N for different block sizes. We can predict the location of the onset of goodput collapse by detailed modeling of TCP slow start and buffer occupancy. Table 1 shows the slow start conges-

tion window sizes versus each packet round trip. For 16 KB blocks, 12 concurrent senders of the largest congestion window of 5864 bytes would require 70368 bytes of buffer, larger than the available buffer of 64 KB per port. Goodput collapse begins after $N = 13$ concurrent senders. The discrepancy of 1 comes from the fact that there is additional "buffer" on the network beyond the packet buffer on the switch, e.g., packets in flight, buffer at the sender machines, etc. According to this logic, goodput collapse should take place according to Equation 3. The equation accurately predicts that for Figure 4, the goodput collapse for 16 KB, 32 KB, and 64 KB blocks begin at 13, 7, and 4 concurrent senders, respectively, and for Figure 3, the goodput collapse is well underway at 2 concurrent senders.

$$N_{initialGoodputCollapse} = \left\lceil \frac{perSenderBuffer}{largestSlowStartCwnd} \right\rceil + 1 \qquad (3)$$

Round trip	16KB blocks	32KB blocks	64KB blocks	128KB blocks
1	1,448	1,448	1,448	1,448
2	2,896	2,896	2,896	2,896
3	5,792	5,792	5,792	5,792
4	5,864	11,584	11,584	11,584
5		10,280	23,168	23,168
6			19,112	46,336
7				36,776

Table 1: TCP slow start congestion window size in bytes versus number of round trips. We verified using `sysctl` that Linux begins at 2x base MSS, which is 1448 bytes.

Takeaway: For small flows, the switch buffer space determines the onset of incast.

Second Order Effects

Figure 4 also suggests the presence of second-order effects not explained by Equations 1 to 3. Equation 3 predicts that goodput collapse for 128 KB blocks should begin at $N = 2$ concurrent senders, while the empirically observed goodput collapse begins at $N = 4$ concurrent senders. It turns out that block sizes of 128 KB represent a transition point from RTO-during-slow-start to more complex modes of behavior.

We repeat the experiment for block size = 128 KB, 256 KB, 512 KB, and 1024 KB. Figure 5 shows the results, which includes several interesting effects.

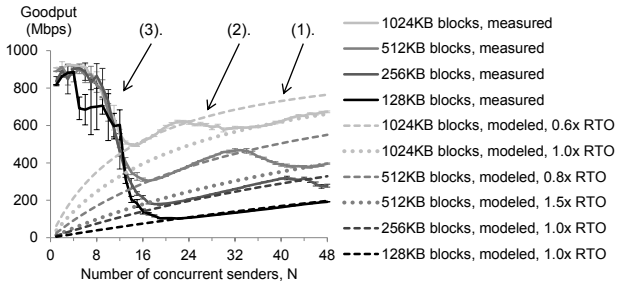


Figure 5: Second-order effects other than RTO during slow start. Measurements done on HP Procurve 5412 switches (64 KB per port). RTO is 200 ms. Error bars represent 95% confidence interval around the average of five repeated measurements. Showing (1) partial RTOs more accurately modeling incast behavior for large blocks, (2) transition between single and multiple partial RTOs, and (3) triple duplicate ACKs causing more gradual, block size–independent onset of incast.

First, for block size = 512 KB and 1024 KB, the goodput immediately after the onset of incast is given by Equation 4. It differs from Equation 2 by the multiplier α for the RTO in the denominator. This α is an empirical constant and represents a behavior that we call partial RTO. What happens is as follows. When RTO takes place, TCP SACK (turned on by default in Linux) allows transmission of further data, until the congestion window can no longer advance due to the lost packet. Hence, the link is idle for a duration of less than the full RTO value. Hence we call this effect partial RTO. For block size = 1024 KB, α is 0.6, and for block size = 512 KB, α is 0.8.

$$Goodput_{beforeIncast} = \frac{blockSize}{\alpha \times RTO + RTT + \frac{blockSize \times N}{linkBandwidth}} \times N \qquad (4)$$

Second, beyond a certain number of concurrent senders, α transitions to something that approximately doubles its initial value (0.6 to 1.0 for block size = 1024 KB, 0.8 to 1.5 for block size = 512 KB). This simply represents that two partial RTOs have occurred.

Third, the goodput collapse for block size = 256 KB, 512 KB, and 1024 KB is more gradual compared with the cliff-like behavior in Figure 4. Further, this gradual goodput collapse has the same slope across different block size. Two factors explain this behavior. First, flows with block size greater than 128 KB have a lot more data to send even after the buffer space is filled with packets sent during slow start (Equation 3 and Table 1). Second, even when the switch drops packets, TCP can sometimes recover. Empirical evidence of this fact exists in Figure 4. There, for block size = 16 KB and N = 13 to 16 concurrent senders, at least one of five repeated measurements manages to get goodput close to 90% of link capacity. Goodput collapse happens for other runs because the packets are dropped in a way that a connection with little additional data to send would observe only a single or double duplicate ACK and would go into RTO soon after. Larger blocks suffer less from this problem because the ongoing data transfers trigger triple duplicate ACK with higher probability. Thus, the connection retransmits, enters congestion avoidance, and avoids RTO. Hence the gradual goodput collapse.

We should point out that SACK semantics are independent of duplicate ACKs, since SACK is layered on top of existing cumulative ACK semantics [23].

Takeaway: Second-order effects include partial RTO due to SACK, multiple partial RTOs, and triple duplicate ACKs causing more gradual onset of incast.

Good Enough Model

Unfortunately, some parts of the model remain qualitative. We admit that the full interaction between triple duplicate ACKs, slow start, and available buffer space requires elaborate treatment far beyond the flow rate and buffer occupancy analysis presented here.

That said, the models here represent the first time we quantitatively explain major features of the incast goodput collapse. Comparable results in related work [28, 25] can be explained by our models also. The analysis allows us to reason about the significance of incast for future big data workloads later in the article.

Incast in Hadoop MapReduce

Hadoop represents an interesting case study of how incast affects application-level behavior. Hadoop is an open source implementation of MapReduce, a distributed computation paradigm that played a key part in popularizing the phrase "big data." Network traffic in Hadoop consists of small flows carrying control packets for various cluster coordination protocols, and larger flows carrying the actual data being processed. Incast potentially affects Hadoop in complex ways. Further, Hadoop may well mask incast behavior, because the network forms only a part of the overall computation and data flow. Our goal for this section is to answer whether incast affects Hadoop, by how much, and under what circumstances.

We perform two sets of experiments. First, we run stand-alone, artificial Hadoop jobs to find out how much incast impacts each component of the MapReduce data flow. Second, we replay a scaled-down, real-life production workload using previously published tools [17] and cluster traces from Facebook, a leading Hadoop user, to understand the extent to which incast affects whole workloads. These experiments take place on the same DETER machines as those in the previous section. We use only the large buffer Procurve switch for these experiments.

Stand-alone jobs

Table 2 lists the Hadoop cluster settings we considered. The actual stand-alone Hadoop jobs are hdfsWrite, hdfsRead, shuffle, and sort. The first three jobs stress one part of the Hadoop I/O pipeline at a time. Sort represents a job with 1-1-1 ratio between read, shuffled, and written data. We implement these jobs by modifying the randomwriter and randomtextwriter examples that are pre-packaged with recent Hadoop distributions. We set the jobs to write, read, shuffle, or sort 20 GB of terasort format data on 20 machines.

EXPERIMENT SETUP

Parameter	Values
Hadoop jobs	hdfsWrite, hdfsRead, shuffle, sort
TCP version	Linux-2.6.28.1, 1ms-min-RTO
Hadoop version	0.18.2, 0.20.2
Switch model	HP Procurve 5412
Number of machines	20 workers and 1 master
fs.inmemory.size.mb	75, 200
io.file.buffer.size	4096, 131072
io.sort.mb	100, 200
io.sort.factor	10, 100
dfs.block.size	67108864, 536870912
dfs.replication	3, 1
mapred.reduce.parallel.copies	5, 20
mapred.child.java.opts	-Xmx200m, -Xmx512M

Table 2: Hadoop parameter values for experiments with stand-alone jobs

The TCP versions are the same as before—standard Linux 2.6.28.1, and modified Linux 2.6.28.1 with `tcp_rto_min` set to 1 ms. We consider Hadoop versions 0.18.2 and 0.20.2. Hadoop 0.18.2 is considered a legacy, basic, but still relatively stable and mature distribution. Hadoop 0.20.2 is a more fully featured distribution that introduces some performance overhead for small jobs [17]. Subsequent Hadoop improvements have appeared on several disjoint branches that are currently being merged, and 0.20.2 represents the last time there was a single mainline Hadoop distribution [30].

The rest of the parameters are detailed Hadoop configuration settings. Tuning these parameters can considerably improve performance, but requires specialist knowledge about the interaction between Hadoop and the cluster environment. The first value for each configuration parameter in Table 2 represents the default setting. The remaining values are tuned values, drawn from a combination of Hadoop sort benchmarking [1], suggestions from enterprise Hadoop vendors [4], and our own experiences. One configuration worth further explaining is `dfs.replication`. It controls the degree of data replication in HDFS. The default setting is threefold data replication to achieve fault tolerance. For use cases constrained by storage capacity, the preferred method is to use HDFS RAID [14], which achieves fault tolerance with 1.4x overhead, much closer to the ideal onefold replication.

RESULTS

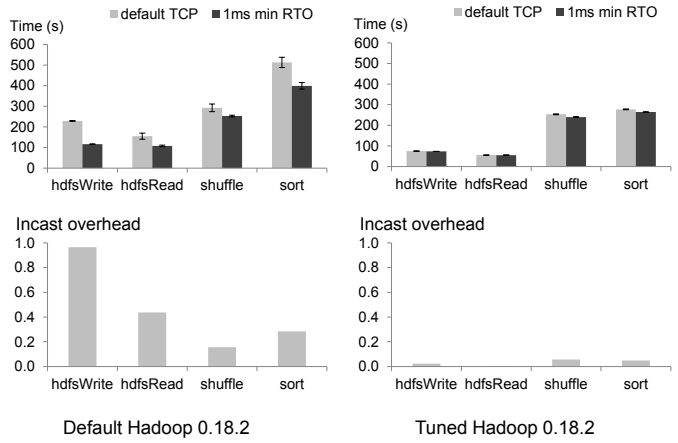


Figure 6: Hadoop stand-alone job completion times and incast overhead. Measurements done on HP Procurve 5412 switches (64 KB per port). The error bars show 95% confidence intervals from 20 repeated measurements. The confidence intervals are not overlapping for both settings.

Figure 6 shows the results for Hadoop 0.18.2. We consider two performance metrics: job completion time and incast overhead. We define incast overhead according to Equation 5, i.e., the difference between job completion time under default and 1 ms-min-RTO TCP, normalized by the job completion time for 1 ms-min-RTO TCP. The default Hadoop has very high incast overhead, while for tuned Hadoop, the incast overhead is barely visible. However, the tuned Hadoop-0.18.2 setting leads to considerably lower job completion times.

$$t = jobCompletionTime$$

$$IncastOverhead = \frac{t_{defaultTCP} - t_{1ms-min-RTO}}{t_{1ms-min-RTO}} \qquad (5)$$

The results illustrate a subtle form of Amdahl's Law, which explains overall improvement to a system when only a part of the system is being improved. Here, the amount of incast overhead depends on how much network data transfers contribute to the overall job completion time. The default Hadoop configurations result in network transfers contributing to a large fraction of the overall job completion time. Thus, incast overhead is clearly visible. Conversely, for tuned Hadoop overall job completion time is already low. Incast overhead is barely visible because the network transfer time is low.

We repeat these measurements on Hadoop 0.20.2. Compared with Hadoop 0.18.2, the more recent version of Hadoop sees a performance improvement for the default configuration. For the optimized configuration, Hadoop 0.20.2 sees performance overhead of around 10 seconds for all four job types. This result is in line with our prior comparisons between Hadoop versions 0.18.2 and 0.20.2 [17]. Unfortunately, 10 seconds is also the performance improvement for using TCP with 1ms-min-RTO. Hence, the performance overhead in Hadoop 0.20.2 masks the benefits of addressing incast.

Takeaway: Incast does affect Hadoop. The performance impact depends on cluster configurations, as well as data and compute patterns in the workload.

Real-life Production Workloads

The results in the above subsection indicate that to find out how much incast *really* affects Hadoop, we must compare the default and 1 ms-min-RTO TCP while replaying real-life production workloads.

Previously, such evaluation capabilities have been exclusive to enterprises that run large-scale production clusters. Recent years have witnessed a slow but steady growth of public knowledge about front-line production workloads [29, 10, 17, 15, 9], as well as emerging tools to replay such workloads in the absence of production data, code, and hardware [17, 16].

WORKLOAD ANALYSIS

We obtained seven production Hadoop workload traces from five companies in social networking, e-commerce, telecommunications, and retail. Among these companies, only Facebook has so far allowed us to release their name and synthetic versions of their workload. We do have permission to share some summary statistics. The full analysis is under publication review.

Several observations are especially relevant to incast. Consider Figure 7, which shows the distribution of per job input, shuffle, and output data for all workloads. First, all workloads are dominated by jobs that involve data sizes of less than 1 GB. For jobs so small, scheduling and coordination overhead dominate job completion time. Therefore, incast will make a difference only if the workload intensity is high enough that Hadoop control packets alone would overwhelm the network. Second, all workloads do contain jobs at the 10s TB or even 100s TB scale. This compels the operators to use Hadoop 0.20.2. This version of Hadoop is the first to incorporate the Hadoop fair scheduler [29]. Without it, the small jobs arriving behind very large jobs would see FIFO head of queue blocking and would suffer wait times of hours or even days. This feature is so critical that cluster operators use it despite the performance overhead for small jobs. Hence, it is likely that in Hadoop 0.20.2, incast will be masked by the performance overhead.

WORKLOAD REPLAY

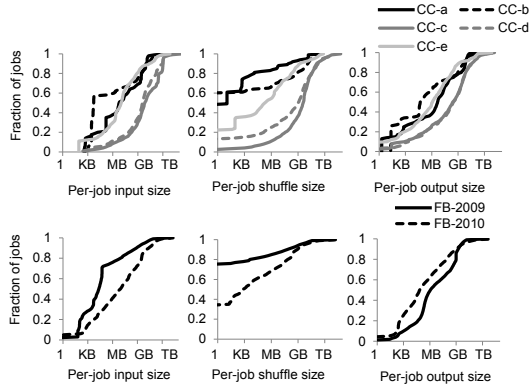


Figure 7: Per job input, shuffle, and output size for each workload. *FB-** workloads come from a six-month cluster trace in 2009 and a 45-day trace in 2010. *CC-** workloads come from traces of up to two months long at various customers of Cloudera, which is a vendor of enterprise Hadoop.

We replay a day-long Facebook 2009 workload on the default and 1 ms-min-RTO versions of TCP. We synthesize this workload using the method in [17]. It captures in a relatively short synthetic workload the representative job submission and computation patterns for the entire six-month trace.

Our measurements confirm the hypothesis earlier. Figure 8 shows the distribution of job completion times. We see that the distribution for 1 ms-min-RTO is 10–20 seconds right-shifted compared with the distribution for default TCP. This is in line with the 10–20 seconds overhead we saw in the workload-level measurements in [17], as well as the stand-alone job measurements earlier in the article. The benefits of addressing incast are completely masked by overhead from other parts of the system.

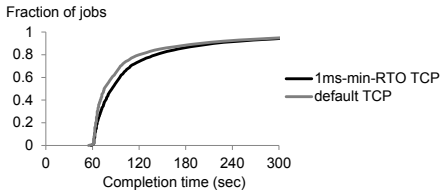


Figure 8: Distribution of job completion times for the *FB-2009* workload.

Figure 9 offers another perspective on workload-level behavior. The graphs show two sequences of 100 jobs, ordered by submission time, i.e., we take snapshots of two continuous sequences of 100 jobs out of the total 6000+ jobs in a day. These graphs indicate the behavior complexity once we look at the entire workload of thousands of jobs and diverse interactions between concurrently running jobs. The 10–20 seconds performance difference on small jobs becomes insignificant noise in the baseline. The few large jobs take significantly longer than the small jobs and stand out visibly from the baseline. For these jobs, there are no clear patterns to the performance of 1 ms-min-RTO versus standard TCP.

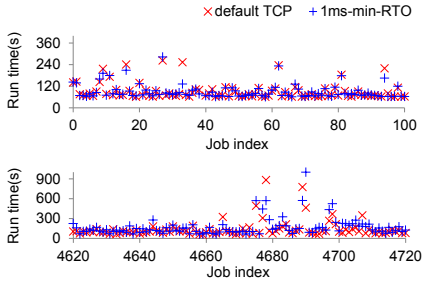


Figure 9: Sequences of job completion times

The Hadoop community is aware of the performance overheads in Hadoop 0.20.2 for small jobs. Subsequent versions partially address these concerns [22]. It would be worthwhile to repeat these experiments once the various active Hadoop code branches merge back into the next mainline Hadoop [30].

Takeaway: Small jobs dominate several production Hadoop workloads. Non-network overhead in present Hadoop versions masks incast behavior for these jobs.

Incast for Future Big Data Workloads

Hadoop is an example of the rising class of big data computing paradigms, which almost always involve some amount of network communications. To understand how incast affects future big data workloads, one needs to appreciate the technology trends that drive the rising prominence of big data, the computational demands that result, and the countless design and mis-design opportunities, as well as the root causes of incast.

We believe that the top technology trends driving the prominence of big data include (1) increasingly easy and economical access to large-scale storage and computation infrastructure [11, 7]; (2) ubiquitous ability to generate, collect, and archive data about both technology systems and the physical world [19]; and (3) growing desire and statistical literacy across many industries to understand and derive value from large datasets [2, 13, 24, 20].

Several data analysis trends emerge, confirmed by the cluster operators who provided the traces in Figure 7:

1. There is increasing desire to do interactive data analysis, as well as streaming analysis. The goal is to have humans with non-specialist skills explore diverse and evolving data sources, and once they discover a way to extract actionable insights, such insights should be updated based on incoming data in a timely and continuous fashion.
2. Bringing such data analytic capability to non-specialists requires high-level computation frameworks built on top of common platforms such as MapReduce. Examples of such frameworks in the Hadoop MapReduce ecosystem include HBase, Hive, Pig, Sqoop, Oozie, and others.
3. Data sizes grow faster than the size per unit cost of storage and computation infrastructure. Hence, efficiently using storage and computational capacity are major concerns.

Incast plays into these trends as follows. The desire for interactive and streaming analysis requires highly responsive systems. The data sizes required for these computations are small compared with those required for computations on historical data. We know that when incast occurs, the RTO penalty is especially severe for small flows. Applications would be potentially forced to either delay the analysis response or give answers based on partial data. Thus, incast could emerge as a barrier for high quality interactive and streaming analysis.

The desire to have non-specialists use big data systems suggests that functionality and usability should be the top design priorities. Incast affects performance, which can be interpreted as a kind of usability. It becomes a priority only after we have a functional system. Also, as our Hadoop experiments demonstrate, performance tuning for multi-layered software stacks would need to confront multiple layers of complexity and overhead.

The need for storage capacity efficiency entails storing compressed data, performing data deduplication, or using RAID instead of data replication to achieve fault tolerance. In such environments, memory locality becomes the top concern, and disk or network locality becomes secondary [8]. If the workload characteristics permit a high level of memory or disk locality, network traffic gets decreased, the application performance increases, and incast becomes less of a concern.

The need for computational capacity efficiency implies that computing infrastructure needs to be more highly utilized. Network demands will thus increase. Consolidating diverse applications and workloads multiplexes many network traffic patterns. Incast will likely occur with greater frequency. Further, additional TCP pathologies may be revealed, such as the similarly phrased TCP outcast problem, which affects link share fairness for large flows [26].

Recommendations

Set TCP minimum RTO to 1 ms.

Future big data workloads likely reveal TCP pathologies other than incast. Incast and similar behavior are fundamentally transport-level problems. It is not resource effective to overhaul the entire TCP protocol, redesign switches, or replace the datacenter network to address a single problem. Setting tcp_rto_min is a configuration parameter change that produces low overhead, is immediately deployable, and, as we hope our experiments show, does no harm inside the datacenter.

Deploy better tracing infrastructure.

It is not yet clear how much incast will impact future big data workloads. This article discusses several contributing factors, but we need further information to determine which factors dominate under what circumstances. Better tracing helps remove the uncertainty. Where possible, such insights should be shared with the general community. We hope the workload comparisons in this article encourage similar, cross-organizational efforts elsewhere.

Apply a scientific design process.

We believe future big data systems demand a departure from some design approaches that emphasize implementation over measurement and validation. The complexity, diversity, scale, and rapid evolution of such systems imply that mis-design opportunities proliferate, redesign costs increase, experiences rapidly become obsolete, and intuitions become hard to develop. Our approach in this article involves performing simplified experiments, developing models based on first principles, empirically validating these models, then connecting the insights to real life by introducing increasing levels of complexity. We hope our experiences tackling the incast problem demonstrate the value of a design process rooted in empirical measurement and evaluation.

Acknowledgments

This research is supported in part by the UC Berkeley AMP Lab (https://amplab.cs.berkeley.edu/sponsors/), and the DARPA- and SRC-funded MuSyC FCRP Multiscale Systems Center. Thank you to Rik Farrow and Sara Alspaugh for proofreading a draft of the article. Thank you also to Keith Sklower for assistance with the DETER Testbed logistics.

References

[1] Apache Hadoop documentation: http://hadoop.apache.org/common/docs/r0.20.2/cluster_setup.html#Configuring+the+Hadoop+Daemons.

[2] Hadoop World 2011 speakers: http://www.hadoopworld.com/speakers/.

[3] IEEE 802.1Qau standard—Congestion notification: http://www.ieee802.org/1/pages/802.1au.html.

[4] Personal communications with Cloudera engineering and support teams.

[5] M. Alizadeh et al. Data center TCP (DCTCP). In SIGCOMM 2010.

[6] M. Alizadeh et al. Data center transport mechanisms: Congestion control theory and IEEE standardization. In Annual Allerton Conference 2008.

[7] Amazon Web Services. Amazon Elastic Compute Cloud (Amazon EC2): http://aws.amazon.com/ec2/.

[8] G. Ananthanarayanan et al. Disk-locality in datacenter computing considered irrelevant. In HotOS 2011.

[9] G. Ananthanarayanan et al. PACMan: Coordinated memory caching for parallel jobs. In NSDI 2012.

[10] G. Ananthanarayanan et al. Scarlett: Coping with skewed content popularity in MapReduce clusters. In EuroSys 2011.

[11] Apache Foundation. Apache Hadoop: http://hadoop.apache.org/.

[12] T. Benzel et al. Design, deployment, and use of the deter testbed. In DETER 2007.

[13] D. Borthakur et al. Apache Hadoop goes realtime at Facebook. In SIGMOD 2011.

[14] D. Borthakur et al. HDFS RAID. Tech talk. Yahoo Developer Network. 2010.

[15] Y. Chen et al. Energy efficiency for large-scale MapReduce workloads with significant interactive analysis. In EuroSys 2012.

[16] Y. Chen et al. SWIM—Statistical workload injector for MapReduce: https://github.com/SWIMProjectUCB/SWIM/wiki.

[17] Y. Chen et al. The case for evaluating MapReduce performance using workload suites. In MASCOTS 2011.

[18] Y. Chen et al. Understanding TCP incast throughput collapse in datacenter networks. In WREN 2009.

[19] EMC and IDC iView. Digital universe: http://www.emc.com/leadership/programs/digital-universe.htm.

[20] M. Isard et al. Quincy: Fair scheduling for distributed computing clusters. In SOSP 2009.

[21] E. Krevat et al. On application-level approaches to avoiding TCP throughput collapse in cluster-based storage systems. In PDSW 2007.

[22] T. Lipcon and Y. Chen. Hadoop and performance. Hadoop World 2011: http://www.hadoopworld.com/session/hadoop-and-performance/.

[23] M. Mathis et al. Request for Comments: 2018—TCP selective acknowledgment options: http://tools.ietf.org/html/rfc2018, 1996.

[24] S. Melnik et al. Dremel: Interactive analysis of Web-scale datasets. In VLDB 2010.

[25] A. Phanishayee et al. Measurement and analysis of TCP throughput collapse in cluster-based storage systems. In FAST 2008.

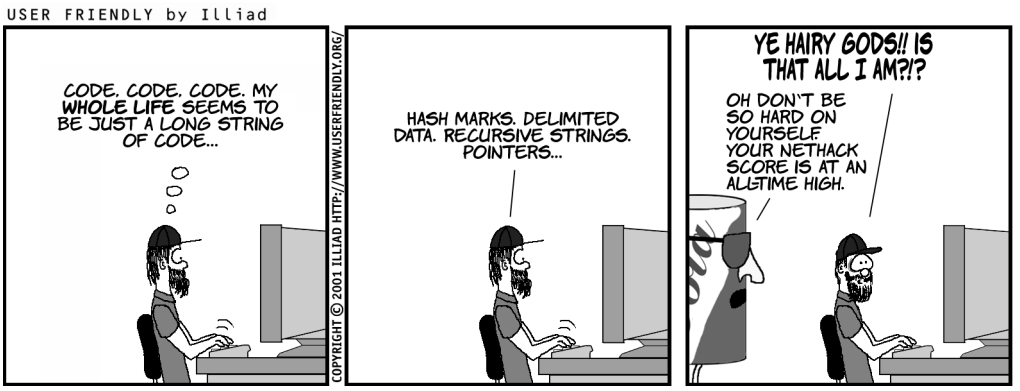
[26] P. Prakash et al. The TCP outcast problem: Exposing throughput unfairness in data center networks. In NSDI 2012.

[27] V. Vasudevan et al. Safe and effective fine-grained TCP retransmissions for datacenter communication. In SIGCOMM 2009.

[28] H. Wu et al. ICTCP: Incast congestion control for TCP in data center networks. In Co-NEXT 2010.

[29] M. Zaharia et al. Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling. In EuroSys 2010.

[30] C. Zedlewski. An update on Apache Hadoop 1.0: http://www.cloudera.com/blog/2012/01/an-update-on-apache-hadoop-1-0/.



An Introduction to HyperDex and the Brave New World of High Performance, Scalable, Consistent, Fault-tolerant Data Stores

ROBERT ESCRIVA, BERNARD WONG, AND EMIN GÜN SIRER

Robert Escriva is a PhD student in computer science at Cornell University. He focuses on building infrastructure services for cloud computing.
escriva@cs.cornell.edu

Bernard Wong is an Assistant Professor in the School of Computer Science at the University of Waterloo. His research interests span distributed systems and networking, with particular emphasis on problems involving decentralized services, self-organizing networks, and distributed storage systems.
bernard.wong@uwaterloo.ca

Emin Gün Sirer is an Associate Professor of Computer Science at Cornell University. He works on infrastructure services for cloud computing and secure operating systems.
egs@systems.cs.cornell.edu

A new generation of data storage systems is now emerging to support high-performance, large-scale Web services whose demands are ill-met by traditional RDBMSes. Dubbed the NoSQL movement, this trend has produced systems characterized by data stores that provide weak consistency guarantees and limit the system interface. We argue that these systems have too aggressively capitulated, that much stronger consistency, availability, and fault-tolerance properties are possible, and, further, that it is possible to provide these properties while offering a rich API, although not as rich as full-blown SQL. We report on a recent system called HyperDex, describe the new techniques it uses to combine strong consistency and fault-tolerance guarantees with high-performance, and go through a scenario to see how the system can be used by real applications.

ACID and BASE

During the golden age of databases, when the canonical database users were banks and other financial institutions, providing strong guarantees of atomicity, consistency, isolation, and durability (ACID) were of paramount concern. More recently, however, the focus of data storage innovation has shifted away from supporting financial transactions to enabling Web services, such as Google, Facebook, and Amazon.com, that need to respond to queries efficiently, scale up to vast numbers of users, and tolerate the server failures that are inescapable at Web scale.

The flagship for this shift away from traditional RDBMS concerns towards properties that are better suited for Web services is a movement called NoSQL. This movement represents a constellation of new data storage systems that forego the traditional ACID guarantees of RDBMSs, along with their SQL interface, for improvements along the dimensions that matter to scalable Web applications. Although the NoSQL name suggests that the removal of SQL is the driving force behind the movement, it is really just the focal point for an overhaul of the storage system interface. For example, rather than having rigid schemas and support for complex search queries, most NoSQL systems have relaxed schemas and favor key-based operations whose implementation can be made scalable and efficient.

Yet the NoSQL movement has, in many ways, tossed the baby out with the bathwater. Most NoSQL systems subscribe to an alternative to ACID called the BASE approach, whose fundamental pillars are Basically Available service, Soft-State, and Eventually Consistent data. It is true that achieving Web scale will require hard tradeoffs between conflicting desires; yet the BASE approach represents a capitulation across *all* fronts. It provides no fault-tolerance guarantee and achieves

no longevity for data, and typical BASE systems struggle to always return up-to-date results even with no failures. The name is catchy, but the resulting systems are quite weak and are useful only to a small niche of applications that can accept best-effort guarantees.

In this article, we provide a brief introduction to HyperDex, a second-generation distributed key-value store that is fast, scalable, strongly consistent, and fault-tolerant. By strongly consistent, we mean that a `get` will always return the latest value placed in the system by a `put`, not just eventually, but always, even during failures and reconfiguration. By fault-tolerant, we mean a system that can tolerate up to `f` failures, whether they are node (server) failures or network partitions affecting up to `f` hosts. And by fast, we mean a system with a streamlined implementation that, on the industry-standard YCSB benchmark, outperforms Cassandra [6] and MongoDB [1], two popular NoSQL systems, by a factor of 2 to 13. And above all, HyperDex supports a new lookup primitive by which objects stored in the system can be recalled by their attributes. Thus HyperDex combines the scalability and high performance properties of NoSQL systems with the consistency and fault-tolerance properties of RDBMSs, while providing a rich API. This unique combination of features is made possible by two novel techniques, *hyperspace hashing* and *value dependent chaining*, that determine the way HyperDex distributes its data.

Hyperspace Hashing

A key-value store, as its name suggests, provides users access to its data through key-based operations, such as `put` and `get`. Most large-scale key-value stores that support horizontal scaling either use a hashing function to map keys to nodes, such as Cassandra [6] and Dynamo [4], or partition the keyspace into contiguous regions that are assigned to different nodes by a centralized coordinator, such as BigTable [3] or HBase [2].

In contrast, HyperDex uses a new object placement method, called *hyperspace hashing*, that takes into account many object attributes when mapping objects to servers. Hyperspace hashing creates a multidimensional Euclidean space, where each dimension corresponds to one searchable attribute, that is, an attribute that may be used as part of a search query. An object's position in this space is specified by its coordinate, which can be determined by hashing the object's searchable attribute values. Objects' schemas are fixed, and different object types necessarily reside in different hyperspaces. Of course, nothing prevents a HyperDex deployment from having multiple spaces with the same hyperspace structure.

For example, a space of objects with "first name," "last name," and "phone number" searchable attributes corresponds to a three-dimensional hyperspace where each dimension corresponds to one attribute in the original object. Such a space is depicted in Figure 1. There are three objects in this space. The circular point is "John Smith" whose phone number is 555-8000. The square point is "John Doe" whose phone number is 555-7000. The diamond point is "Jim Bob" whose phone number is 555-2000. Anyone named "John" *must* map to somewhere in the plane labeled "John." Similarly, anyone with the last name "Smith" *must* map to somewhere within the plane labeled "Smith." Naturally, all people named "John Smith" must map to somewhere along the line where these two planes intersect.

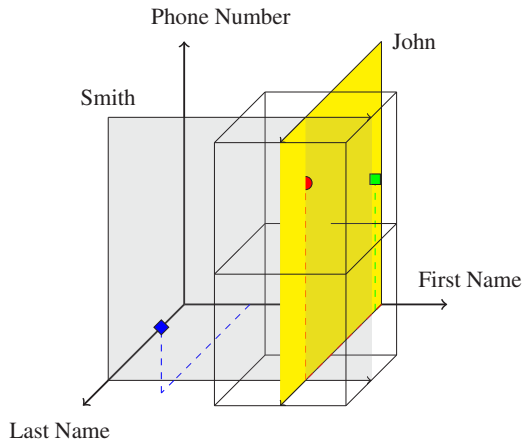


Figure 1: Simple hyperspace hashing in three dimensions. Each plane passes through all points corresponding to a specified query. Together the planes represent a line through all phone numbers for a given first name and last name pair. The cubes show two of the eight zones in this hyperspace each of which is handled by different servers.

For each space, HyperDex tessellates the hyperspace into disjoint pieces called *zones*, and assigns nodes (servers) to each zone. Figure 1 shows two of these assignments. Notice that the line for "John Smith" only intersects two out of the eight assignments. Consequently, performing a search for all phone numbers of "John Smith" requires contacting only two nodes. Furthermore, the search could be made more specific by restricting it to all people named "John Smith" whose phone number falls between 555-5000 and 555-9999. Such a search contacts only one out of the eight servers in this hypothetical deployment.

This simple object-mapping technique is not without pitfalls. Objects with many attributes translate to hyperspaces with many dimensions. The volume of the resulting hyperspace grows exponentially in the number of dimensions/attributes. A naïve approach would be to restrict the number of searchable attributes, and thus the size of the hyperspace. Such a technique limits the utility of hyperspace hashing. HyperDex avoids exponential growth of the hyperspace while maintaining the utility of hyperspace hashing by creating multiple independent and smaller hyperspaces, called *subspaces*. A large object may be represented in constant-size hyperspaces, the number of which is linear to the number of searchable attributes in the object. Here, HyperDex trades storage efficiency for search efficiency.

An additional pitfall with naïve hyperspace hashing is that key lookups would be equivalent to single attribute searches, which would likely be inefficient compared to key lookups in other key-value stores. Fortunately, using subspace partitioning, it is trivial to construct a subspace containing *just* the key of the object. This ensures that a `get` operation will always contact exactly one server in this subspace.

Value-Dependent Chaining

In addition to providing good performance and scalability, a distributed storage system must also provide fault tolerance. Much like other distributed storage systems, HyperDex achieves fault tolerance through data replication. However, HyperDex's use of hyperspace hashing and subspace partitioning introduce additional challenges, as the two features in combination force the same object to be stored

at more than one server, which in turn presents problems of consistency between these replicas. As the location of an object in each subspace can change with every object update, the location of the replicas will also change. The replication scheme must therefore be able to manage replica sets that change frequently.

One replication approach, used in NoSQL systems that preceded HyperDex, would be to use an eventually consistent update mechanism. Such a mechanism would allow each replica to accept updates, and at a later point, the updates would be propagated to the rest of the replicas. However, changes to the replica set from multiple concurrent updates could result in inconsistency across subspaces. This type of inconsistency can accumulate over time and result in significant divergence between the contents of different subspaces. Furthermore, detecting such divergences is non-trivial and likely involves some form of all-to-all communication.

Instead, HyperDex introduces a new replication protocol called *value-dependent chaining* that efficiently provides total ordering on replica set updates. In value-dependent chaining, each update is propagated to the affected server nodes through a well-defined linear pipeline. Updates flow down the chain, while acknowledgments flow back up the chain. The head of the chain is the node responsible for that object's key, called a point leader. Because all value dependent chains for the same object have the same point leader, all updates to that object can be fully ordered with respect to each other. Node failures lead to broken chains, which are fixed automatically by shifting all nodes below the point of breakage up a spot and adding a new spare node at the tail of the chain to restore the desired level of fault tolerance. Failures of the point leader are handled the same way, with the backup point leader becoming the new node responsible for that zone. This linear ordering ensures the invariant that there is never any confusion about which nodes have seen the most fresh updates; consequently, there is no need for expensive mechanisms such as voting, leader election, or quorum writes.

Value-dependent chains also provide an additional property for free: all key operations are strongly consistent. The same chaining mechanisms that consistently update the replica set ensure consistent updates to the objects, without any overhead beyond what is required to maintain consistency of the replica set.

Tutorial

HyperDex has been fully implemented and is freely available for download. It includes all of the features we have described in this article. It is also being actively developed, with a small but growing development community that is eager to add developer-friendly features and additional language bindings. In this section, we will illustrate how a simple phonebook application uses HyperDex as its storage back-end.

Creating a HyperDex Space

A phonebook application needs to, at a bare minimum, keep track of a person's first name, last name, and phone number. In order to distinguish unique users, it might assign to each a user ID. We can instruct HyperDex to create a suitable space for holding such objects with the following command:

```
hyperdex-coordinator-control
        --host 127.0.0.1 --port 6970
        add-space << EOF
```

```
space phonebook
dimensions username, first, last
        phone (int64)
key username auto 1 3
subspace first, last, phone auto 3 3
EOF
```

This command creates a new space called phonebook that stores objects with the following four searchable attributes: username, first name, last name, and phone number. In this example, the space creation command instructs HyperDex to create a 1-dimensional subspace for the key, and a 3-dimensional subspace for the remaining attributes.

The replication level is specified by the "1 3" and "3 3" parameters at the end of the key and subspace line. This instructs HyperDex to divide the key subspace into 21 zones and the subspace for the remaining attributes into 23 zones, and to replicate each zone on to three nodes. As a general rule, a HyperDex administrator should configure HyperDex to not have significantly more zones per subspace than the number of nodes in the deployment.

Basic Operations

With a hyperspace defined, our phonebook application can connect to HyperDex and begin issuing basic get and put requests. We illustrate the HyperDex API using our Python client.

```
import hyperclient
c = hyperclient.Client('127.0.0.1', 1234)
```

This code snippet instructs the client bindings to talk to the HyperDex controller and retrieve the current HyperDex configuration. The controller ensures that the clients always receive the most up-to-date configuration. If the configuration changes, say, due to failures, the servers will detect that a client is operating with an out-of-date configuration and instruct it to retry with the updated HyperDex configuration.

Now that our phone application has created a client, it can insert objects in the system by issuing `put` requests:

```
c.put('phonebook', 'jsmith1',
        {'first': 'John', 'last': 'Smith', 'phone': 6075551024})
True
c.put('phonebook', 'jd',
        {'first': 'John', 'last': 'Doe', 'phone': 6075557878})
True
```

The client determines the unique location in the hyperspace for an object, contacts the servers responsible, and issues the `put` request to these servers. Similarly, our phone application can retrieve the jsmith1 object by issuing a get request.

```
c.get('phonebook', 'jsmith1')
{'first': 'John', 'last': 'Smith', 'phone': 6075551024}
```

Our phone application can also use HyperDex's search primitive to retrieve objects based on one or more secondary attributes.

```
[x for x in c.search('phonebook',
          {'first': 'John', 'last': 'Smith', 'phone': 6075551024})]
[{'first': 'John', 'last': 'Smith',
          'phone': 6075551024,
          'username': 'jsmith1'}]
[x for x in c.search('phonebook', {'first': 'John'})]
[{'first': 'John', 'last': 'Smith', 'phone': 6075551024, 'username': 'jsmith1'},
{'first': 'John', 'last': 'Doe', 'phone': 6075557878, 'username': 'jd'}]
[x for x in c.search('phonebook', {'last': 'Smith'})]
[{'first': 'John', 'last': 'Smith', 'phone': 6075551024, 'username': 'jsmith1'}]
[x for x in c.search('phonebook', {'last': 'Doe'})]
[{'first': 'John', 'last': 'Doe', 'phone': 6075557878, 'username': 'jd'}]
```

Should the user decide to remove "John Doe" from his/her phonebook, the phonebook application can remove the object by issuing a `delete` request:

```
c.delete('phonebook', 'jd')
True
[x for x in c.search('phonebook', {'first': 'John'})]
[{'first': 'John', 'last': 'Smith', 'phone': 6075551024, 'username': 'jsmith1'}]
```

Finally, if the user wants to locate everyone named "John Smith" from Ithaca (area code 607), the phonebook application can issue the following range query to HyperDex:

```
[x for x in c.search('phonebook',
          {'last': 'Smith', 'phone': (6070000000, 6080000000)})]
[{'first': 'John', 'last': 'Smith', 'phone': 6075551024, 'username': 'jsmith1'}]
```

Atomic Read-Modify-Write Operations

HyperDex offers several atomic read-modify-write operations which are impossible to implement in key-value stores with weaker consistency guarantees. These operations, in turn, enable concurrent applications that would otherwise be impossible to implement correctly using non-atomic operations. For instance, using standard `get` and `put` operations, an application cannot ensure that its operations will not be interleaved with operations from other clients.

The canonical example for needing atomic read-modify-write operations involves two clients who are both trying to update a salary field. One is trying to deduct taxes—let's assume that they are hard-working academics being taxed at the maximum rate of 36%. The other client is trying to add a $1500 teaching award to the yearly salary. So one client will be doing:

```
v1=get(salary), v1 = v1 - 0.36*v1; put(salary, v1)
```

while the other client will be doing:

```
v2=get(salary), v2 += 1500; put(salary, v2)
```

where v1 and v2 are variables local to each client. Since these get and put operations can be interleaved in any order, it is possible for the clients to succeed (so both the deduction and the raise are issued) and yet for the salary to not reflect the

results! If the sequence is get from client1, get from client2, put from client2, put from client1, the raise will be overwritten—a most undesirable outcome.

Atomic read-modify-write operations provide a solution to this problem. Such operations are guaranteed to execute without being interrupted by or interleaved with any other operation.

The word "atomic" is often associated with poor performance; however, HyperDex's atomic operations are inexpensive and virtually indistinguishable from a put, thanks to the use of value-dependent chains. The head of each object's value-dependent chain is in a unique position to locally compute the result of the atomic operation and, should it succeed, pass the operation down the chain as a normal `put`. Should the operation fail, the remainder of the value-dependent chain does not need to be involved at all.

HyperDex supports a few different atomic instructions, the most general of which is a conditional_put. A conditional_put performs the specified `put` operation if and only if the value being updated matches a specified condition.

Continuing with the sample phonebook application, consider extending the application for use in login authentication. The phonebook table must then be extended to include a password attribute. Intuitively, a user should only be able to change his/her password when it matches the password that he/she used to log in. The phonebook application can do this by using conditional_put:

```
c.conditional_put('phonebook', 'jsmith',
          {'password': 'currentpassword'},
          {'password': 'newpassword'})
True
c.get('phonebook', 'jsmith1')
{'first': 'John', 'last': 'Smith', 'phone': 6075552048,
           'password': 'newpassword'}
```

Although this toy example omits certain implementation details relating to secure password storage, it is clear that the `conditional_put` operation enables behavior that is otherwise impossible to achieve with normal `get` and `put` operations. Any attempt to change the password without providing the previous password will fail:

```
c.conditional_put('phonebook', 'jsmith',
          {'password': 'wrongpassword'},
          {'password': 'newpassword'})
False
```

As expected, the `conditional_put` failed because the password is not, in fact, "wrongpassword".

HyperDex offers additional atomic operations. In many applications, the clients will want to increment or decrement a numerical field in the style of Google +1 and Reddit up/down votes. While implementing this is trivial with conditional_put, the implementation may require multiple attempts as the conditional_put operations fail in the face of contention. Atomic increment operations, in contrast, will not fail spuriously, and do not require the user to have retrieved the old value before starting the operation.

We further extend our sample phonebook application to track the number of times each user's information is viewed by adding a "lookups" attribute. The phonebook

application can consistently manage this counter using the atomic_increment operation:

```
c.atomic_increment('phonebook', 'jsmith1', {'lookups': 1})
True
```

The atomic increment is as inexpensive as a put operation. This enables our application to log each lookup quickly and efficiently.

Asynchronous Operations

So far, we submitted synchronous operations to the key-value store, where the client had just a single outstanding request and waited patiently for that request to complete. In high-throughput applications, clients may have a batch of operations they want to perform on the key-value store. The standard practice in such cases is to issue asynchronous operations, where the client does not immediately wait for each individual operation to complete. HyperDex has a very versatile interface for supporting this use case.

Asynchronous operations allow a single client library to achieve higher throughput by submitting multiple simultaneous requests in parallel. Each asynchronous operation returns a small token that identifies the outstanding asynchronous operation, which can then be used by the client, if and when needed, to wait for the completion of selected asynchronous operations.

Every operation we've covered so far in the tutorials (e.g., get) has a corresponding version prefixed with async_ for performing that operation asynchronously. The basic pattern of usage for asynchronous operations is to initiate the asynchronous operation, do some work, perhaps issue more operations, and then wait for selected asynchronous operations to complete. This enables the application to continue to do other work while HyperDex performs the requested operations.

Here's how we could insert the "jsmith" user asynchronously:

```
d = c.async_put('phonebook', 'jsmith1',
       {'first': 'John', 'last': 'Smith',}
        'phone': 6075551024})
d
<hyperclient.DeferredInsert object at 0x7f2bbc3252d8>
do_work()
d.wait()
True
d = c.async_get('phonebook', 'jsmith1')
d.wait()
{'first': 'John', 'last': 'Smith', 'phone': 6075551024}
```

Notice that the return value of the first d.wait() is True. This is the same return value that would have come from performing c.put(...), except the client was free to do other computations while HyperDex servers were processing the put request. Similarly, the second asynchronous operation, async_get, queues up the request on the servers, frees the client to perform other work, and yields its results only when wait is called.

This allows for powerful applications. For instance, it is possible to issue thousands of requests and then wait for each one in turn without having to serialize the round trips to the server. Note that HyperDex may choose to execute concurrent

asynchronous operations in any order. It's up to the programmer to order requests by calling wait appropriately.

Fault Tolerance

HyperDex provides a strong fault-tolerance guarantee to its clients. Anywhere during the preceding tutorial, feel free to kill off up to two of the nodes in the system. You will be able to continue the tutorial, as the value-dependent chains will detect the failures and route around them. If you bring up new nodes, they will be integrated into the chains seamlessly by the coordinator. The particular fault-tolerance level `f`, which determines the number of simultaneous failures a space can withstand, is entirely up to the application to determine. Of course, there are tradeoffs; while a large `f` will yield a more robust system, it will also increase operation latencies, and the improvement in actual reliability is subject to diminishing returns. The critical issue here is that this tradeoff is not part of the HyperDex substrate but is left up to applications to determine.

Performance

In an accompanying report [5], we carefully quantify HyperDex's performance using the industry-standard YCSB benchmark against Cassandra and MongoDB. While a similar performance study is beyond the scope of this introduction to HyperDex, we will report the major takeaway: HyperDex is very fast. It is approximately 2 to 13 times faster than the fastest of the other two NoSQL systems. There are two reasons for this huge gap in performance, which is even more striking because the other two systems are left in their preferred configurations, where they provide weak fault-tolerance and consistency guarantees. First, hyperspace hashing provides an enormous speedup for search-oriented operations. There is a qualitative difference between systems that enumerate objects by iterating through the keyspace and HyperDex, which can use the hyperspace to efficiently pick the desired items, so the 13x improvement could have been even higher if the benchmark's dataset had been larger. Second, HyperDex has a more streamlined implementation that is 2 to 4 times faster than Cassandra and MongoDB even at traditional `get/put` workloads. The precise details of the comparisons are in the technical report, and the beauty of open source is that there is tangible proof in a public repository that anyone can trivially check out and execute.

Summary

The emergence of large-scale Web applications has significantly altered the trajectory of distributed storage systems. From the radically different requirements of Web applications, NoSQL systems have emerged to fill the gap left by traditional databases. Early NoSQL systems used simple techniques, such as consistent hashing and parallel RPCs, to distribute their data, and thus were not able to make nuanced tradeoffs between desirable properties. In this article we presented HyperDex, a new high-performance key-value store that provides strong consistency guarantees, fault-tolerance against failures whose maximum size can be bounded, and high performance coupled with a rich API. These techniques are made possible through the use of hyperspace hashing and value-dependent chaining, two novel techniques for laying out and managing data. We hope that HyperDex, with its strong consistency and fault-tolerance guarantees, high performance, and rich API, will enable a new class of applications that were not served well by existing NoSQL systems.

Acknowledgments

We would like to thank the HyperDex open source community for their contributions, feedback, and support. In addition, we would like to highlight the extensive contributions of Pawel Loj and By Zhang, who have submitted substantial functionality to improve HyperDex.

References

[1] 10gen, Inc.: http://www.mongodb.org, accessed November 29, 2011.

[2] Apache Software Foundation: http://hbase.apache.org, accessed November 29, 2011.

[3] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Michael Burrows, Tushar Chandra, Andrew Fikes, and Robert Gruber, "Bigtable: A Distributed Storage System for Structured Data," *Proceedings of OSDI*, November 2006, pp. 205–218.

[4] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshmanx, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels, "Dynamo: Amazon's Highly Available Key-value Store," *Proceedings of SOSP*, October 2007, pp. 205–220.

[5] Robert Escriva, Bernard Wong, and Emin Gün Sirer, "HyperDex: A Distributed, Searchable Key-Value Store for Cloud Computing," Computer Science Department, Cornell University Technical Report, December 2011.

[6] Avinash Lakshman and Prashant Malik, "Cassandra—A Decentralized Structured Storage System," *Proceedings of LADIS*, October 2009.

Nathan Milford on Cassandra

An Interview

RIK FARROW

Nathan Milford (@NathanMilford) is the US Operations Manager at Outbrain. Nathan is interested in large data projects, scalable architectures, and open source. In his spare time, he is a photographer and practitioner of Jiu Jitsu and Muay Thai. You can follow his blog at http://blog.milford.io/.
nathan@milford.io

Doug Hughes introduced me to Nathan Milford when he learned that I was looking for someone who could talk about his experience using Cassandra. Before talking with Nathan, I read his excellent slide deck about working with Cassandra [1], and watched part of his presentation of these slides [2] from the Cassandra NYC conference last December.

After a short phone conversation over the noise in his datacenter, Nathan agreed to continue our talk by email.

Rik: Could you tell us a little about what Outbrain does, to provide us with the background we need to understand why you chose to use Cassandra?

Nathan: I'll just hit you with what my marketing team would have me say:

> Outbrain is the leading content discovery platform, helping publishers, brands, and agencies reach a highly engaged audience through distribution on leading media sites. Outbrain works with publishers like CNN, Fox News, Hearst, Rolling Stone, and MSNBC as well as brands and agencies, including American Express, P&G, General Electric, Media Contacts, and Starcom to increase site traffic and generate new revenue through customized links to recommended content.

In short, we're a content discovery and recommendation engine. We've got dozens of paid and organic recommendation algorithms that dig into our Hadoop, Solr, Cassandra, and other clusters and return, not only other content that is like what you're reading, but other content that will likely be interesting to you.

Rik: That does sound interesting, but could you provide more detail?

Nathan: Gladly. We use Cassandra as a persistent cache of calculated recommendations.

The (somewhat simplified) flow for our operation goes something like this:

- A user opens up a an article on, say, CNN.com.
- Our widget loads from a CDN and pings one of our three datacenters with the site and document IDs.
- A bevy of Tomcat instances behind HAProxy grab the document info, then query Memcached looking for pre-calculated recommendations for that document.
- If Memcached doesn't have it, Tomcat will ping another app we call the CacheWarmer.
- If it is a new document, the CacheWarmer will send a request into ActiveMQ (a commonly used queue and message broker) to have various offline processes

crawl, index, and calculate the recommendations for it. This process can (depending on the algorithms involved) hit Solr, Hadoop/Hive, MogileFS, MySQL, Cassandra, and/or a bunch of other internal processes that our brilliant R&D teams have concocted.

- The calculated recommendation data is then put into Memcached, where it will eventually expire, and into Cassandra, where it lives for much longer but also will eventually expire, thanks to Cassandra's TTL feature.
- If it is a document we know about, we hit up Cassandra for it and float it into Memcached.

We're also building other Cassandra clusters for other uses. We have a large document mapping table in MySQL that is essentially a key/value store, and a good fit for Cassandra's data model.

Before we started using data stores other than MySQL, we had a single-master MySQL setup with slaves distributed across datacenters. Since we're read-heavy, it makes sense. However, data and traffic keep on growing, and fixing replication issues and managing a brittle topology requires more and more attention.

When not all of your data needs the features MySQL offers, you come to a place where you weigh the advantages of federating your data out into appropriate data stores and having to manage a menagerie of newfangled systems versus fitting all your data into MySQL and dealing with the feature overhead and keeping a system everyone already knows.

It's not for everyone, but we chose to use the menagerie.

Rik: Why did you choose Cassandra over other NewSQL databases? Were others in the running?

Nathan: We started using TokyoTyrant on SSDs, but at the time the project had a small community, the developer was not always responsive, and it was a bit unpolished operationally. It was not crash-safe, and managing replication was a challenge sometimes, in that the mechanism was pretty basic.

We looked into HBase, but we were turned off by the HDFS append patches you needed to mess with at the time (it has since gotten better and more reliable). Also, we wanted something that would reduce operational complexity, so running multiple Hadoop clusters just to run HBase on top of it as well as keeping the clusters in sync seemed like a lot of work.

Cassandra hit the sweet spot for performance and operational complexity. Dealing with replication across multiple datacenters is pretty trivial.

The biggest difficulty is getting people to model data for it properly and not treat it like MySQL. Once you model the data and have a query plan that suits it, Cassandra is pretty hands-off from an operational perspective.

We've been using Cassandra in production since version 0.5.0 (1.1 was just released). We've had some rough patches, but nothing wildly discouraging, and, for the most part, it just works.

Since 0.5.0 we've gotten a SQL-like query language called CQL, JDBC drivers, rolling upgrades, live schema management, encryption, compression, TTLs, secondary indexes, distributed counters, pluggable everything, performance parity between reads and writes, and a wildly long list of other great work by all the committers and community.

I think one of Cassandra's strongest attributes is the Cassandra community, which is very open and accepting of even people with the smallest, most trivial use cases. You can even get commercial support from the guys at DataStax, all of whom are pretty sharp folks.

You also have large players like Twitter and Netflix using it. [Ed.: Netflix is moving away from Oracle to use Cassandra exclusively [3, 4].] Netflix actually showed how linearly it scales by doing a stress test scaling from 0 to 288 nodes in EC2 [5].

Rik: When we talked earlier, you mentioned having a LAN party where everyone got Cassandra set up and running in less than 20 minutes. Is it really that easy?

Nathan: Yes. I am one of the organizers of the NYC Cassandra Meetup group along with Ed Capriolo, Jake Luciani, Levon Lloyd, and Eric Tamme. Ed had a wonderful idea where we'd have everyone bring a laptop (Windows, Mac, and Linux) with a recent JVM. We divided people into three groups, told them to plug into a different switch representing a different "datacenter," and had them install the Cassandra binary package.

The hardest part was herding everyone into the respective areas and then onto the network. It took ~30–40 minutes to get everyone set up with the right network settings and maybe 10 minutes after that to get everyone on the cluster. Shortly after that we were inserting a key in "New York" and watching it replicate to "France" and "Tokyo" [6].

Cassandra is pretty complex, but the majority of that complexity exists to keep you from having to worry about its complexity.

I do a talk on how easy it is and what a boon it is to not have to deal with replication and repair and other administrative junk.

Rik: How did you size up your requirements, that is, the number of Cassandra nodes you needed for your application?

Nathan: We were not very scientific about it when we started 2–3 years ago. We do 30 billion impressions a month, about a billion a day. You can do all the speculation and math and planning in the world, but at that scale, you just need to put traffic on it and let it sink or swim.

For the most part we found it to be a good swimmer.

Ultimately, the first iteration of our cluster was just some spare nodes. Over time our data, our traffic, and Cassandra's performance profile changed and we migrated to new hardware while we played with different file systems, disk configs, row and key caching, heap sizes, garbage collectors, etc.

Our current hardware spec is in the slide deck [1], but we have more nodes and are running 1.0.7 now.

Rik: How difficult is it to add new nodes?

Nathan: It is not difficult really, but it is a reasonably manual process. You need to recalculate your ring [7], then start up the new nodes with the correct token (which defines what part of the ring they own), then move each node's token and let them shuffle the data.

It is all a background process and is only limited by your bandwidth. If you have a multi-datacenter cluster, and slow transport between them . . . well, you'll just have to wait.

Cassandra is not for every use-case and certainly not for every type of data, but all in all, I'm very happy we went with it. It fits a nice niche in our environment and the community around it is a joy to participate in.

References

[1] Nathan Milford's slide deck on Cassandra: http://www.slideshare.net/nmilford/cassandra-for-sysadmins.

[2] Milford's video on using Cassandra: http://blog.milford.io/2012/01/cassandra-nyc-2011-talk-on-youtube/.

[3] Adrian Cockcroft on Migrating from Oracle to Cassandra: http://www.slideshare.net/adrianco/migrating-netflix-from-oracle-to-global-cassandra.

[4] Interview with Adrian Cockcroft in *;login:*: https://www.usenix.org/publications/login/february-2012/netflix-heads-clouds-interview-adrian-cockcroft.

[5] Adrian Cockcroft and Denis Sheahan, "Benchmarking Cassandra Scalability on AWS": http://techblog.netflix.com/2011/11/benchmarking-cassandra-scalability-on.html.

[6] Cassandra/LAN party: http://www.datastax.com/dev/blog/cassandra-nyc-lan-party; http://www.edwardcapriolo.com/roller/edwardcapriolo/entry/cassandranyc_nosql_lan_party_was.

[7] Rebalancing after adding a node: http://blog.milford.io/2011/05/in-which-i-discourse-on-java-bloat-and-cassandra-node-balancing/.

When Disasters Collide

A Many-Fanged Tale of Woe, Coincidence, Patience, and Stress

DOUG HUGHES AND NATHAN OLLA

Doug Hughes is the manager for the infrastructure team at D. E. Shaw Research, LLC., in Manhattan. He is a past LOPSA board member and was the LISA '11 conference co-chair. Doug fell into system administration accidentally, after acquiring a BE in Computer Engineering, and decided that it suited him.
doug@will.to

Nathan Olla is a system administrator at D. E. Shaw Research, LLC., where his focus is on data availability and backup solutions. Nathan found system administration by volunteering over a decade ago to support something called "Linux" while working at Dell.

Never in my career have I experienced as many things go wrong as I did back in late January/early February of 2012. Strangely, not one of them was in any way related to any of the others; they just happened at the same time! Harrowing doesn't begin to cover our feelings in the midst of this maelstrom. I rode out hurricane Opal, as it ripped through Alabama, with less stress. We had four unrelated near-disasters in the span of about four days, and nearly lost 800 TB of data. This is that story, plus a few semi-interesting lessons.

Issue 1

Our problems started with the network. We have monitors in place (e.g., SmokePing [1]) that monitor latency on our wide area network. Our nominal latency is about 11 ms between our primary office site and our primary datacenter site. The network between them is an OC-12 which is an optical, leased line of about 640 Mbps traversing several carriers. We also have a 100 Mbps Internet link which can act as a backup connection, via IPSEC VPN, when the OC-12 is down. Occasionally, one of the OC-12 WAN providers has a maintenance or a minor service-impacting event, and that latency will jump to around 60 ms as the traffic takes an alternate path through the provider's networks, maintaining the same bandwidth. When this happens, the latency consistency (jitter) is relatively consistent, meaning that all packets have a round trip time (RTT) tightly clustered around 60 ms.

An example of a major event would be equipment failure on either end; this takes down the OC-12 connection entirely. On the day of the incident, we saw latency in the 20 ms range but very jittery, which is more characteristic of our backup link on 100 Mbps service. This did not appear to be a standard provider-internal path reroute. We managed to confirm this theory fairly quickly by running iperf [2] between the two sites, yielding a paltry 50 Mbps instead of our more normal 300+ Mbps. We opened a ticket with the primary provider, the one to whom we send the monthly check.

You may notice that the rerouted OC-12 path has much higher latency than the 100 megabit VPN path. We believe this is because the rerouted path is being directed through a distant state before coming back to NYC, although it is difficult to say for certain. There are several carriers who service the end-to-end circuit as we know it. We've seen cases where the local loop can add a significant amount of latency under pathological conditions. The high latency during provider maintenances is puzzling and under active investigation.

A further troubleshooting difficulty with our WAN situation is that we don't actually manage the routers containing the OC-12 linecards. We have a gigabit handoff with a partner organization who manages them for us. This means that when we have a non-trivial difficulty we need to engage this partner organization and the carrier organizations in a coordinated fashion. Such was this case. The carrier checked and re-checked the lines and could not find anything wrong. We've had instances where the OC-12 failback took more than 24 hours, so at first we assumed that this was one of those cases; a particular sub-carrier has been known to close tickets claiming nothing is wrong many times previously. In turn, we've repeatedly had to reopen and insist that "No, 60 milliseconds really isn't normal, there' s definitely something wrong. Please check again!" For this failure, after three days without OC-12, we were starting to worry. The worry was less directed than it might have been because of three other incidents that all happened while this one was occurring, but more on that later.

After many exchanges between carrier and partner, using loopback tests on both sides to prove the circuit was up, we isolated the problem to the datacenter side of the circuit. Upon further examination, we found that the link between the carrier-provided OC-12 equipment and the partner-provided OC-12 router, only about 15 meters of fiber, was the likely culprit. On the carrier side was a long range (LR) single-mode fiber-optical transceiver, and on the partner side was an intermediate range (IR) fiber-optic transceiver. Going by the specifications, the dB losses for both were mismatched. Also, IR transceivers have a range of about 15 km and LR have a range of about 40 km. It might be considered good luck that it managed to work for four years! We suggested that both sides adopt short range (SR) transceivers for the 15 meter distance to avoid overpowering the optical receivers on the other end. Neither organization had SR optics on hand. Both would have had to order the parts, leading to 2–3 more weeks of running on the 100 megabit backup link, which was clearly unacceptable. The local carrier, however, did have an IR transceiver on hand. So, after four days and replacing the LR transceiver we were up and running again on the primary link!

As a side note, the users were relatively unaffected by all of this because of some appliances we use on both ends of our WAN link. Among other useful features, the SilverPeak appliances that we use provide dynamic compression and optimization, network memory, and QoS:

- **Network memory** is the capacity to collect streams of data, store the patterns that haven't been seen before on disk buffers on both ends (keyed with a strong checksum algorithm), and, when that pattern is recognized on the sender side, send the checksum key instead of the entire data stream. The remote side, seeing the key on the WAN interface, pulls the data pattern corresponding to the key from the local disk and feeds it to the requester over the LAN interface. This works in both directions.
- **Compression and optimization** encompasses both TCP header compression and a standard off-the-shelf data compression algorithm to compress the data portion of the packets. It also performs optimization on TCP patterns such as window and buffer sizes, retransmit rates, etc., to get better usage of the available bandwidth. Obviously, compressibility of the data in question is an important factor, but can result in significant bandwidth reductions.

- **QoS** allows one to prioritize:
  - What streams get priority over others, by protocol (ssh, http, nfs, general TCP traffic, etc.)
  - What minimum bandwidth a given protocol is guaranteed in the face of contention
  - What maximum bandwidth rate a protocol can consume
  - Whether a stream is compressed or not (e.g., it makes no sense to try to compress ssh or https, nor to use network memory for either. If it were repeatable data, it wouldn't be secure.).

For us, QoS was the most important feature for running over the lower bandwidth connection, followed by network memory, which meant that patterns that had been requested before and were able to be fed to the requester at gigabit speeds. We prioritize ssh and VNC above all others, so interactive sessions were only mildly impacted. Not a single complaint ticket was originated.

Ponderables:

- Make sure that your backup links work; do periodic testing.
- Having QoS is very useful.
- Make sure that your optical transceivers are appropriately matched and appropriate for the distance of the fiber run.
- Your carrier(s) is/are likely to have escalation protocols. Make sure you know what these are for emergencies.

Issue 2

The second issue struck in the first day of the OC-12 outage. One of our 160 TB (raw) backup storage servers running ZFS lost knowledge of a group of about eight disks. We configured these servers with 6 by 10 disk Raidz2 stripes. This meant that two of the stripes were down two disks. A third disk failure in either of the Raidz2 sets (down two disks already) would mean a very large amount of lost backup data, because ZFS stripes blocks across all six of the Raidz2 stripe sets.

We've experienced peculiarities with the RAID controllers in this system before. Each controller puts a label on every disk to indicate what logical unit (LUN) the disk is a part of. This logical unit can be a simple pass-through, or one of several different hardware RAID configurations. A pass-through is the same as taking a disk that sits behind the controller, stamping a label on it, and passing it through as a logical unit to the host OS (1:1); RAID LUNs are 1:n. The logical unit also includes information such as whether read or write caching should be enabled, among other things. This was the first time a controller had lost its capricious little memory of eight disks at once. Suspiciously, they were all adjacent disks in the chassis, but we could find no subsequent hardware problems, and there were other disks on the same controller that had no issues.

So we did what any normal organization would do under such circumstances. We tried pulling and reseating the disks. It didn't help. We tried checking the SAS cables, which also didn't help. We tried power cycling the entire server. That didn't appear to help either. It turns out that the disks were physically there in the RAID controller view, but had disappeared from the OS view. We ended up having to re-add the RAID controller logical unit (LUN) labels onto the disks to make them show up to the OS. Once we did that, they were visible again in zpool status, but in a very odd way. If you remember my previous article covering ZFS (undetected

bit flips), you'll be familiar with the way that zpool status looks. This is how it was showing up to us on that day (as representative of a single Raidz2 stripe with two disks from the amnesia set):

```
c1t24d0    ONLINE   0    0     0
c1t25d0    ONLINE   0    0     0
c1t26d0    ONLINE   0    0     0
11422982192057997398    FAULTED    0    0    0    was /dev/dsk/c2t0d0s0
3080189289823161725     FAULTED    0    0    0    was /dev/dsk/c2t1d0s0
c2t12d0    ONLINE   0    0     0
```

Instead of a controller number, there's a big integer. Okay, this isn't a huge problem. We can also see that the right column contains information about what the disk was. Hey, that's handy! Or is it? No amount of zpool replace, add, or remove was able to deal with these disks. The zpool commands for removing and replacing disks told us the disks didn't exist, while the commands for adding told us that they were already a member of the pool! We even tried a 14+ hour scrub of the entire zpool, but that didn't recover anything any better.

It turns out that each of the labels for what the disk used to be (right column) also existed elsewhere in the status output for the 82 disks in the system. In other words, the disk mapping was FUBAR when the disks were lost. Unfortunately, we didn't discover this until after I had tried some creative means of fixing these disks. We thought about the best way to recover these disks and were using the name from the right column before realizing that it was a conflict. I used dd to overwrite the disk label to, hopefully, make ZFS forget about the disks so that they could be replaced. This is a bit dangerous, but at the time we felt we had nothing to lose since the disks were already unused by, and unusable to, ZFS. But this made ZFS forget about eight other legitimate, functional, and otherwise healthy disks in the system! Several stripes were now down three disks (below critical) and one was down five disks! Hello, worst case…

Fortunately, we were able to revert this situation by carefully reconstructing which disks were what. After many hours of trying things, we removed the eight offending disks from the system (remember, they were adjacent) and rebooted. The file system was still unrecoverable in this state. We re-inserted the disks and used the Solaris format –e command followed by label to force an EFI label onto the accidentally demolished disks. At this point, ZFS managed to detect the correct info from the remaining disks and we could bring the zpool online. It reconstructed the broken label on the dd'd disks and we were back to where we had been the day before. After running another zpool scrub we determined that dd had blown away data blocks in two legitimate files that were contained in snapshots. In other words, we didn't lose anything of primary value, just a copy of snapshotted copies of two old versions of files. It could have been a lot worse.

The problem remained that the original eight missing disks still had a broken ZFS label on them. We had to very carefully identify them since they did not have a usable /dev/dsk identification that could be used accurately at the system level. Using the long integer was also a failure; ZFS did not like that. Remember, we could neither remove them from ZFS knowledge, because they were not part of the current pool, nor could we add them to the pool, because their label said that they were part of an existing pool. Even using zpool commands with –f would not work.

We had to find a disk that wasn't already thought to be somewhere else in the system. Using zpool information correlated with ls /dev/dsk and the MegaCLI (which manages LUNs) command to map the single disk LUNs to disk numbers in the system, we were able to find a single, spare disk that was not being mapped to a LUN. This was painstaking work. Using this disk, we were able to replace one of the disks with the large integer identifier. Then we deleted that logical unit, re-created it, ran devfsadm to map it to a cXtXdX designation, added it to the zpool as a spare, and were able to use it to replace the next drive, and so on, one after another, 2 TB at a time, until we were done. Only by fixing the labels one at a time were we able to repair the system. Other than losing two old versions of files in snapshots, we made a full recovery, but we were perilously close to losing all of the backups on this system.

Along the way, we had some hilarious exchanges via Jabber, like this classic:

```
|2012-01-30T14:31:34|1|to|N---|that's curious -- grub prompt
|2012-01-30T14:31:48|1|from|N---|hrm.. let's reset again
|2012-01-30T14:32:58|1|from|N---|the 4 flash drives are there
|2012-01-30T14:33:13|1|from|N---|(or 1, divided into 4 chunks, anyway)
|2012-01-30T14:33:15|1|to|N---|they were before as well
|2012-01-30T14:33:40|1|to|N---|great, my console shrunk down to a teeny
window and can't be resized
|2012-01-30T14:34:43|1|from|N---|heh
|2012-01-30T14:34:50|1|from|N---|well, it's at grub again
…
|2012-01-30T18:08:17|1|to|N---|zpool import -Fn + slaughter a chicken?
|2012-01-30T18:08:27|1|to|N---|i guess that won't help
```

Ponderables:

- When you see "was" in ZFS output, make sure that it doesn't match something that is already there.
- You can use dd to wipe out a disk label, and ZFS will still be able to repair it under many circumstances. (This saved our bacon!)
- Use RAID cards that do not require creating an explicit single disk logical unit. RAID cards that do implicit pass-through of disks that are not part of hardware RAID set logical units seem to work much better and be less confusing overall. Certainly they are less trouble to set up on an 80-disk system.
- Check your sanity by double-checking your mapping for apparently missing disks. They may not be what you think they are.
- Locate lights are helpful. Most RAID CLI software supports this.
- If you can't replace a disk with itself because of some intermediate mapping SNAFU, try replacing it with a spare and working step-wise through the disks to get back to normal.

Issue 3

Sometime on the second day of the network outage, we were dismayed to find that an NFS monitor had triggered on one of our two primary application servers (which are exact copies of each other). These servers are not very sizeable machines, but they have enough memory to keep the ZFS adaptive replacement cache (ARC [3]) for all of the actively served NFS files in memory. They also have a level-2 ARC (L2ARC) MLC (multi-level cell) SSD disk for holding more than twice

RAM for any overflow. They also have 10-gigabit network interface cards (NIC) tuned to reduce the number of interrupts when something like a Python application launches and requests 10,000 accesses to NFS paths. Even when un-cached on the client, the server can feed the requested files (or lack thereof) back in ones or tens of microseconds instead of the typical ~10 milliseconds required if they had to be fetched from spinning media.

One of these two servers had gone unreachable. Unfortunately, this meant that roughly half of our cluster machines were stuck on new NFS application requests, owing to two factors:

1. Linux does not support failover to an alternate server for read-only NFS shares. Our NFS clients are all Linux.
2. Independent servers without shared storage cannot be set up as an NFS cluster in Solaris 10.

We started diagnostics by grabbing the serial console and checking for anything egregious. Unfortunately, it was non-responsive. So we issued a remote power cycle command through the lights out management (LOM) interface. It displayed the BIOS flash screen, passed the memory self-test, and as soon as it reached the part where it interacts with the RAID card BIOS, the server got stuck. We tried another reset and it got stuck in the same place. Clearly, something was wrong there.

Deductively, we suspected that the RAID card had gone bad, so we started to swap it with a RAID card from another machine of the same type. Both machines are engineered well enough that this is an operation that can be done without removing the servers from the rack. This took about 20 minutes, because we had to make sure that the other machine was in a suitable state of disuse.

Unfortunately, after the cards were swapped, and we powered the stuck server on, it got stuck in the same place! Unusual...What else could it be? We theorized that perhaps one of the SSD disks that we use for boot and ZFS intent log (ZIL) had gone bad in a particular way; perhaps it was hanging the SATA bus or controller with resets? So we removed the first of the two mirrored SSDs and did another reset. It got stuck in the same spot. We tried again, after pushing the first SSD back in and removing the second. Eureka! Somehow one of the two disks had failed in such a way that it *had* hung the bus, and removing it allowed the machine to boot normally.

Unfortunately, this had resulted in about 2–3 hours of downtime for quite a number of machines during primary work hours.

Ponderables:

- Make sure you have spares readily on hand, even for RAID cards.
- Having machines that allow for swapping RAID cards, CPUs, RAM, etc., without total removal from the rack is useful for important application servers. Had we not had this, it would have easily added another 30 minutes to the procedures.
- SSDs are still relatively new technology and can have unusual failure modes. Try swapping them before the RAID card.

Issue 4

This was the big one. We nearly lost 640 TB of primary storage! Unfortunately, this article has already gotten somewhat long, so we'll leave this as a tantalizing cliffhanger for the next issue!

References

[1] SmokePing: http://oss.oetiker.ch/smokeping/.

[2] Iperf: http://iperf.sourceforge.net.

[3] ARC: http://en.wikipedia.org/wiki/Adaptive_replacement_cache.

Practical Perl Tools

Rainy Days and Undocumented APIs Always Get Me Down

DAVID N. BLANK-EDELMAN

David N. Blank-Edelman is the director of technology at the Northeastern University College of Computer and Information Science and the author of the O'Reilly book *Automating System Administration with Perl* (the second edition of the Otter book), available at purveyors of fine dead trees everywhere. He has spent the past 24+ years as a system/network administrator in large multi-platform environments, including Brandeis University, Cambridge Technology Group, and the MIT Media Laboratory. He was the program chair of the LISA '05 conference and one of the LISA '06 Invited Talks co-chairs. David is honored to have been the recipient of the 2009 SAGE Outstanding Achievement Award and to serve on the USENIX Board of Directors beginning in June of 2010.
dnb@ccs.neu.edu

It seems like everyone talks about the weather, but few people code about it. It's not that I'm particularly enamored by weather (I go along with the Phantom Tollbooth quote, "I'm the Whether Man, not the Weather Man, for after all it's more important to know whether there will be weather than what the weather will be."), but I think it makes a lovely trampoline from which to explore a few of the more prevalent kinds of Web services/APIs you may encounter. For these demonstrations I'm going to stick to free or close to free services (at least for a personal level of queries). There are some commercial weather data providers who are exceptionally miserly with their data or force you to sign a EULA the size of my arm; they will be conspicuously absent from this column.

One quick note for my international readers: The goal of this column is to demonstrate how to bring Perl to bear to work with these kinds of APIs, not the specific services or APIs themselves. If any of these services fail to cover your particular geographical area, it is possible you can find one that does and use the same techniques to query it. It's not that I don't care deeply and passionately about the weather where you live; it is just that it is easier for me to show code that I can validate by looking out my window.

Weather Provided as XML

One thing that most of these services have in common is that they like to return data in a structured XML format. I thought I would start our exploration by looking at a service that returns a really simple XML document. Before I show this example to you, I have to admit we're going to be a bit naughty. The following example will query a service that doesn't really have a documented API for this purpose and is almost certainly not supported in this context. And even though there is a Perl module available to use this service (though we're going to do it by hand in this column), I can't recommend you use it for anything besides educational/demonstration purposes.

So who provides this API-less service that we're going to use in such a transgressive manner? Google. I realize this is a bit of a surprise given how important APIs are to them, but this is not a separate official service to them. Google provides weather data as part of their ability to customize your Google home page (iGoogle, sigh) with a weather gadget. There is also a small amount of information on how their weather data is represented in a document about customizing their toolbar. Hopefully, this information gives you a sense of just how much in the wilderness we'll be when we attempt to use this service.

That being said, the actual work is really easy. If you make an HTTP GET request to a URL of this form:

```
http://www.google.com/ig/api?weather={some place}
```

it will return an XML document like this (I used Boston, MA, as the place and reformatted the reply for easier reading):

```
<?xml version="1.0"?>
<xml_api_reply version="1">
   <weather module_id="0" tab_id="0" mobile_row="0" mobile_zipped="1"
   row="0" section="0">
      <forecast_information>
         <city data="Boston, MA"/>
         <postal_code data="Boston MA"/>
         <latitude_e6 data=""/>
         <longitude_e6 data=""/>
         <forecast_date data="2012-03-29"/>
         <current_date_time data="2012-03-29 16:54:00 +0000"/>
         <unit_system data="US"/>
      </forecast_information>
      <current_conditions>
         <condition data="Overcast"/>
         <temp_f data="42"/>
         <temp_c data="6"/>
         <humidity data="Humidity: 76%"/>
         <icon data="/ig/images/weather/cloudy.gif"/>
         <wind_condition data="Wind: N at 9 mph"/>
      </current_conditions>
      <forecast_conditions>
         <day_of_week data="Thu"/>
         <low data="34"/>
         <high data="48"/>
         <icon data="/ig/images/weather/rain.gif"/>
         <condition data="Showers"/>
      </forecast_conditions>
      <forecast_conditions>
         <day_of_week data="Fri"/>
         <low data="37"/>
         <high data="50"/>
         <icon data="/ig/images/weather/sunny.gif"/>
         <condition data="Clear"/>
      </forecast_conditions>
      <forecast_conditions>
         <day_of_week data="Sat"/>
         <low data="30"/>
         <high data="45"/>
         <icon data="/ig/images/weather/chance_of_rain.gif"/>
         <condition data="Chance of Rain"/>
      </forecast_conditions>
      <forecast_conditions>
         <day_of_week data="Sun"/>
         <low data="34"/>
```

```
        <high data="55"/>
        <icon data="/ig/images/weather/chance_of_rain.gif"/>
        <condition data="Chance of Rain"/>
      </forecast_conditions>
   </weather>
</xml_api_reply>
```

Now, let's grab the data using Perl and parse it. Since this is really simple XML and our use of this data is straightforward, we can turn to the tremendously helpful XML::Simple module to parse the data. Although XML::Simple can parse the data, it can't fetch it from Google. For that we'll use LWP::Simple, which provides a get() function that will retrieve data for a given URL. Here's the code:

```
use strict;
use LWP::Simple;
use XML::Simple;

my $xml = XMLin( get('http://www.google.com/ig/api?weather=Boston+MA'),
   ValueAttr => ['data'] );

print "Current conditions: "
   . $xml->{weather}->{current_conditions}->{condition} . " "
   . $xml->{weather}->{current_conditions}->{temp_f} . " F\n";

foreach my $day ( @{ $xml->{weather}->{forecast_conditions} } ) {
   print $day->{day_of_week} . ': '
      . $day->{condition} . ' '
      . $day->{high} . '/'
      . $day->{low} . "\n";
}
```

XML::Simple's XMLin function gets called to parse the data retrieved by LWP::Simple. We use the defaults for it with one exception to make our life easier. If you take a look at the sample XML document above, you'll see lines such as:

```
<condition data="Overcast"/>
<temp_f data="42"/>
<temp_c data="6"/>
<humidity data="Humidity: 76%"/>
```

where the elements don't actually hold the data; the attributes of those elements do. By default, XML::Simple will place those attributes into their own separate hashes with the name of the attribute as the key. This means we would ordinarily get a data structure that looks like this excerpt:

```
'current_conditions' => HASH(0x7f8032f32b80)
  'condition' => HASH(0x7f8032f332b8)
    'data' => 'Overcast'
  'humidity' => HASH(0x7f8032f2da90)
    'data' => 'Humidity: 73%'
  'icon' => HASH(0x7f8032f2db20)
    'data' => '/ig/images/weather/cloudy.gif'
  'temp_c' => HASH(0x7f8032f333d8)
    'data' => 6
  'temp_f' => HASH(0x7f8032f33348)
    'data' => 42
```

```
'wind_condition' => HASH(0x7f8032f2dbb0)
  'data' => 'Wind: N at 9 mph'
```

It would be much more pleasant if we could just eliminate the need for a separate sub-hash to hold the values, and instead get something like:

```
'current_conditions' => HASH(0x7fc001733058)
   'condition' => 'Overcast'
   'humidity' => 'Humidity: 73%'
   'icon' => '/ig/images/weather/cloudy.gif'
   'temp_c' => 6
   'temp_f' => 42
   'wind_condition' => 'Wind: N at 9 mph'
```

and indeed, that's what the ValueAttr option to XMLin() does for us in one swell foop. Now you get some sense of why I tend to be pretty effusive in my praise of XML::Simple.

Weather Provided as an RSS Feed

The second kind of weather service I'd like to explore with you is one I introduced in a column back in 2006. There are services that provide weather data to you as RSS feeds. RSS is better known as a blog-related standard, so you may not have encountered it much except as internal plumbing found largely behind the scenes. Wikipedia's got the following lovely description:

> RSS (originally RDF Site Summary, often dubbed Really Simple Syndication) is a family of Web feed formats used to publish frequently updated works—such as blog entries, news headlines, audio, and video—in a standardized format. An RSS document (which is called a "feed," "Web feed," or "channel") includes full or summarized text, plus metadata such as publishing dates and authorship.
>
> RSS feeds benefit publishers by letting them syndicate content automatically. A standardized XML file format allows the information to be published once and viewed by many different programs. They benefit readers who want to subscribe to timely updates from favorite websites or to aggregate feeds from many sites into one place.

The RSS format spec has gone through a number of revisions, but all of them are represented using XML. We could use a generic XML parser to deal with it (as you'll see in the next section in this column), but in this case it is a little easier to use a dedicated RSS module called XML:RSS::Parser to parse the data. There is also an XML::RSS module available that I would normally use, but it doesn't seem to (at least in my experience) play nicely with the slightly customized RSS feed we're going to consume in this section.

Yahoo! is probably the most popular provider that makes RSS feeds for weather available, so we'll use them for the code example for this section. Querying Yahoo! for weather for a US location is as easy as requesting the RSS feed for that location's zip code using a URL:

```
http://xml.weather.yahoo.com/forecastrss?p={zipcode here}
```

Even though that query format works, it is deprecated; instead, Yahoo! now wants you to do this instead:

```
http://weather.yahooapis.com/forecastrss?w={WOEID}
```

In the second format above, you provide a WOEID in the URL using the w parameter. Yahoo! created the WOEID, or "Where On Earth ID," to be a unique identifier for any place on the planet. It's a cooler system than you might expect (so cool Twitter decided to adopt it). More details on it can be found at this Yahoo! URL: http://developer.yahoo.com/geo/geoplanet/guide/concepts.html.

So where do you get the WOEID for a particular place? Yahoo! suggests the easiest way to do so is to search for that place at http://weather.yahoo.com. The resulting URL for that place's weather page will end in the WOEID for that place. For example, if I search for Boston, MA, the URL for the page that is returned has this URL:

```
http://weather.yahoo.com/united-states/massachusetts/boston-2367105/
```

If the idea that you have to type in each place you would want to query into a search box in a browser seems a bit, ehemm, manual to you (and I certainly hope it does to regular readers of this column), Yahoo! provides a place-to-WOEID query service available. It's simple to use, but I think it is out of the scope of this column. For more details, please see http://developer.yahoo.com/geo/geoplanet/.

So let's get back to the task at hand and see how to use XML::RSS::Parser to deal with data from Yahoo!'s RSS-based weather service. XML::RSS::Parser doesn't actually fetch the data from either of the two Yahoo! RSS feed URLs above, so we'll again use LWP::Simple's get() function. Putting all of these pieces together, we get sample code that looks like this:

```
use strict;
use LWP::Simple;
use XML::RSS::Parser;

my $parser = XML::RSS::Parser->new;

# Yahoo! uses a custom namespace for their data
$parser->register_ns_prefix( 'yweather',
   'http://xml.weather.yahoo.com/ns/rss/1.0' );

# 2367105 is the WOEID for Boston, MA
my $feed = $parser->parse_string(
   get('http://weather.yahooapis.com/forecastrss?w=2367105') );

print "Current Conditions: "
   . $feed->query('//yweather:condition/@yweather:text') . " "
   . $feed->query('//yweather:condition/@yweather:temp') . " F\n";

my (@forecasts) = $feed->query('//yweather:forecast');
foreach my $day (@forecasts) {
   print $day->query('@yweather:day') . ': '
      . $day->query('@yweather:text') . ' '
      . $day->query('@yweather:high') . '/'
      . $day->query('@yweather:low') . "\n";
}
```

Most of the code above is pretty straightforward, with one exception. The lines that include code like this might be a bit curious:

```
$feed->query('//yweather:condition/@yweather:text')
```

One of XML:RSS::Parser's strengths (which it gets from the Class::XPath module it uses) is that it provides an XPath-like/lite query language. XPath provides a terse but elegant syntax for finding elements in an XML document. I like it a great deal, so much so that I suspect you can look for a future column on just XPath. In the meantime, let me explain that the code above returns the yweather:text attribute from an XML element with the name "yweather:condition" found anywhere in the document ('//' means anywhere starting from the root element).

The line later in our code that says:

```
my (@forecasts) = $feed->query('//yweather:forecast');
```

is performing a similar query, this time requesting all of the elements called yweather:forecast. We iterate over each of the elements returned by that query, and for each element we request the attributes we want to display:

```
foreach my $day (@forecasts) {
   print $day->query('@yweather:day') . ': '
      . $day->query('@yweather:text') . " "
      . $day->query('@yweather:high') . "/"
      . $day->query('@yweather:low') . "\n";
}
```

Weather Provided as JSON

For our final example, I though it would be good to change up the format we're processing even though XML is by far the most prevalent format being used to provide weather data. But XML itself is starting to get stiff competition from another data interchange format when it comes to Web services these days. The competitor is JSON, made popular because AJAXy things are tending to use it more and more. To understand a bit about JSON, I want to quote verbatim from the json.org Web site:

> JSON (JavaScript Object Notation) is a lightweight data-interchange format. It is easy for humans to read and write. It is easy for machines to parse and generate. It is based on a subset of the JavaScript Programming Language, Standard ECMA-262 3rd Edition—December 1999. JSON is a text format that is completely language independent but uses conventions that are familiar to programmers of the C-family of languages, including C, C++, C#, Java, JavaScript, Perl, Python, and many others. These properties make JSON an ideal data-interchange language.
>
> JSON is built on two structures:
>
> - A collection of name/value pairs. In various languages, this is realized as an object, record, struct, dictionary, hash table, keyed list, or associative array.
> - An ordered list of values. In most languages, this is realized as an array, vector, list, or sequence.
>
> These are universal data structures. Virtually all modern programming languages support them in one form or another. It makes sense that a data format that is interchangeable with programming languages also be based on these structures.

If you've dealt at all with YAML, you will have very little problem coping with JSON (YAML proponents claim it is a superset of JSON). We explored working

with JSON back in the June 2008 *;login:* column, so you may want to take a quick look at that column if you'd like more information on how to work with it from Perl.

For this demonstration, we're going to use the Web service from Weather Underground (wunderground.com) that does charge money for their service if used over a certain amount. The free plan offers you 500 calls per day, 10 calls per minute—more than sufficient for the needs of this column. If you want to go higher, the next tier (5000 calls per day, 100 per minute) is only $20US per month, nothing outrageous.

One quick aside before we actually get into coding against their API: our previous data provider, Yahoo! actually has a "secret" (i.e., not directly documented in their Weather section, as far as I can tell) JSON API available. I think one undocumented API per column is more than enough so I'm going to just mention it exists (use "forecastjson" in the URL instead of "forecastrss") and move on.

To use the Weather Underground API, you need to sign up for an API key. With that key, you can construct your query URL. In the examples below, I've replaced my personal API key with YOURAPIKEY.

Unlike the previous services we've seen, Weather Underground lets you request different "features" from the service by adding keywords to the part of the URL you would normally associate with the path to the resource. For example, if I wanted to retrieve just the current conditions for a place, I would use a URL that began:

```
http://api.wunderground.com/api/YOURAPIKEY/conditions ...
```

If I wanted to duplicate what we've received from the other services by requesting both the current conditions and the forecast, the URL would begin with:

```
http://api.wunderground.com/api/YOURAPIKEY/conditions/forecast ...
```

After the features part of the URL, you provide the location and an indication of the format you'd like back. Here's the complete URL we'd use to get back the current conditions and forecast for Boston as a JSON document:

```
http://api.wunderground.com/api/YOURAPIKEY/conditions/forecast/q/MA/Boston.json
```

Based on our previous examples, you can probably guess what our sample code will look like. The main difference is we'll be feeding the results of our get() to the JSON module's from_json function. This converts the JSON documented into a Perl data structure, along the lines of this (I've heavily excerpted below because the data you get back is pretty voluminous):

```
0  HASH(0x7fd2bad12020)
   'current_observation' => HASH(0x7fd2bad11f18)
      'relative_humidity' => '79%'
      'solarradiation' => 109
      'station_id' => 'KMAWINTH1'
      'temp_c' => 6.1
      'temp_f' => 42.9
      'temperature_string' => '42.9 F (6.1 C)'
      'visibility_km' => 16.1
      'visibility_mi' => 10.0
      'weather' => 'Overcast'
      'wind_degrees' => 47
      'wind_dir' => 'NE'
```

```
    'wind_gust_kph' => 16.1
    'wind_gust_mph' => 10.0
  'forecast' => HASH(0x7fd2baad6758)
    'simpleforecast' => HASH(0x7fd2bad00850)
      'forecastday' => ARRAY(0x7fd2badfcf08)
        0  HASH(0x7fd2badfcfc8)
          'avehumidity' => 67
          'avewind' => HASH(0x7fd2badfdd48)
            'degrees' => 34
            'dir' => 'NE'
            'kph' => 10
            'mph' => 6
          'conditions' => 'Rain Showers'
```

The trickiest part is simply finding the right parts of the data structure to display. Here's our last piece of sample code:

```
use strict;
use LWP::Simple;
use JSON;

my $weather = from_json(

   get('http://api.wunderground.com/api/YOURAPIKEY/conditions/forecast/q/
   MA/Boston.json'
   )
);

print 'Current conditions: '
   . $weather->{current_observation}->{weather} . " "
   . $weather->{current_observation}->{temp_f} . " F\n";

foreach my $day ( @{
     $weather->{forecast}->{simpleforecast}->{forecastday} } ) {
        print $day->{date}->{weekday_short} . ": "
            . $day->{conditions} . ' '
            . $day->{high}->{fahrenheit} . '/'
            . $day->{low}->{fahrenheit} . "\n";
}
```

To end this column, let me show you the current output of the previous code so you can feel a bit better about the weather near you:

```
Current conditions: Overcast 42.9 F
Thu: Rain Showers 46/36
Fri: Clear 48/36
Sat: Chance of Rain 43/30
Sun: Chance of Rain 52/32
```

Take care, and I'll see you next time.

Becoming a Master Collector

DAVID BEAZLEY

David Beazley is an open source developer and author of the *Python Essential Reference* (4th Edition, Addison-Wesley, 2009). He is also known as the creator of Swig (http://www.swig.org) and Python Lex-Yacc (http://www.dabeaz.com/ply.html). He is based in Chicago, where he also teaches a variety of Python courses.
dave@dabeaz.com

As a Python programmer, you know that lists, sets, and dictionaries are useful for collecting data. For example, you use a list whenever you want to store data and keep it in order:

```
>>> names = ['Dave', 'Paula', 'Thomas', 'Lewis']
>>>
```

If you simply want a collection of unique items and don't care about the order, you can make a set:

```
>>> colors = set(['red','blue','green','purple','yellow'])
>>>
```

You use a dictionary whenever you want to make key-value lookup tables:

```
>>> prices = { 'AAPL' : 613.20, 'ACME' : 71.23, 'IBM' : 174.11 }
>>> prices['AAPL']
613.20
>>>
```

Using just these three primitives, you can build just about any other data structure in the known universe. However, why would you? In this article, we reach into Python's collections library and look at some of the tools it provides for manipulating collections of data. If you're like me, these will quickly become a part of your day-to-day programming.

Tabulating Data

How many times have you ever needed to tabulate data or build a histogram? For example, suppose you want to tabulate and count all of the IP addresses that made requests on your Web site from a server log such as this:

```
78.192.56.97 - - [15/Mar/2012:01:50:37 -0500] "GET /ply/ HTTP/1.1" 200 11875
69.237.118.150 - - [15/Mar/2012:01:51:52 -0500] "GET /ply/ply.html HTTP/1.1"
200 107623
69.237.118.150 - - [15/Mar/2012:01:51:57 -0500] "GET /ply/example.html
HTTP/1.1" 200 2393
91.35.214.71 - - [15/Mar/2012:01:52:13 -0500] "GET /ply/ HTTP/1.1" 200 11875
91.35.214.71 - - [15/Mar/2012:01:52:13 -0500] "GET /favicon.ico HTTP/1.1" 404
369
```

You might be inclined to write a small fragment of code using a Python dictionary, like this:

```
hits_by_ipaddr = {}
for line in open("access-log"):
        fields = line.split()
        ipaddr = fields[0]
        if ipaddr in hits_by_ipaddr:
           hits_by_ipaddr[ipaddr] += 1
        else:
           hits_by_ipaddr[ipaddr] = 1
```

Although this code "works," it's also a bit clunky. For example, you have to add a special check for first initialization (otherwise the attempt to increment the count will fail with a `KeyError` on first access). On top of that, after you have populated the dictionary, you will probably want to do some further analysis. For example, maybe you want to print a table showing the 25 most common IP addresses in descending order:

```
popular_ips = sorted(hits_by_ipaddr,
        key=lambda x: hits_by_ipaddr[x],
        reverse=True)

for ipaddr in popular_ips[:25]:
        print("%5d: %s" % (hits_by_ipaddr[ipaddr],ipaddr))
```

As output, this will produce a table such as this:

```
 1096: 78.192.56.97
 1040: 206.15.64.54
  473: 212.85.154.246
  226: 89.215.101.39
  209: 212.85.154.254
  185: 82.226.112.70
  180: 78.192.56.101
...
```

Although this code is relatively easy to write, you still need to think about it a bit—especially the tricky sort with the `lambda`. However, you can avoid all of this if you simply use Counter objects from the collections module. Here is a much simplified version of the same code:

```
from collections import Counter

hits_by_ipaddr = Counter()
for line in open("access-log"):
        fields = line.split()
        ipaddr = fields[0]
        hits_by_ipaddr[ipaddr] += 1

for ipaddr, count in hits_by_ipaddr.most_common(25):
        print("%5d: %s" % (count, ipaddr))
```

First added to Python 2.7, Counter objects are perfectly suited for tabulation. They automatically take care of initializing elements on first access. Not only that, they provide useful methods such as most_common([n]) that return the *n* most common items. However, this is really only scratching the surface.

If you want, counters can be automatically initialized from iterables. For example, let's make letter counts from strings:

```
>>> a = Counter("Hello")
>>> b = Counter("World")
>>> a
Counter({'l': 2, 'H': 1, 'e': 1, 'o': 1})
>>> b
Counter({'d': 1, 'r': 1, 'o': 1, 'W': 1, 'l': 1})
>>>
```

Or, if you're inclined and a bit more sophisticated, you can populate a counter from a generator expression:

```
>>> f = open("access-log")
>>> hits_by_ipaddr = Counter(line.split()[0] for line in f)
>>> hits_by_ipaddr['78.192.56.97']
1096
>>>
```

You can also do math with counters:

```
>>> a + b   # Adds counts together
Counter({'l': 3, 'o': 2, 'e': 1, 'd': 1, 'H': 1, 'r': 1, 'W': 1})
>>> a - b   # Takes away counts in b
Counter({'H': 1, 'e': 1, 'l': 1})
>>> b - a   # Takes away counts in a
Counter({'r': 1, 'd': 1, 'W': 1})
>>> a & b   # Minimum counts
Counter({'l': 1, 'o': 1})
>>> a | b   # Maximum counts
Counter({'l': 2, 'e': 1, 'd': 1, 'H': 1, 'o': 1, 'r': 1, 'W': 1})
>>>
```

Adding and subtracting counts are also available in-place using update() and subtract methods, respectively. For example:

```
>>> a = Counter("Hello")
>>> a
Counter({'l': 2, 'H': 1, 'e': 1, 'o': 1})
>>> a.update("World")
>>> a
Counter({'l': 3, 'o': 2, 'e': 1, 'd': 1, 'H': 1, 'r': 1, 'W': 1})
>>>
```

Using some of these techniques, we can refine our script to process an entire directory of log files:

```
from collections import Counter
from glob import glob

hits_by_ipaddr = Counter()

logfiles = glob("*.log")
for filename in logfiles:
        f = open(filename)
```

```
        hits_by_ipaddr.update(line.split()[0] for line in f)
        f.close()

for ipaddr, count in hits_by_ipaddr.most_common(25):
        print("%5d: %s" % (count, ipaddr))
```

By now, hopefully, you've gotten the idea that Counter objects are the way to go for tabulation. Frankly, they're one of my favorite new additions to Python.

Dictionaries with Multiple Values

Normally, dictionaries map a single key to a single value. However, a common question that sometimes arises is how you map a key to multiple values. Naturally, the solution is to map a key to a list or set. For example, suppose you wanted to make a dictionary that mapped URLs to all of the unique IP addresses that accessed it. Here is some code that would do it:

```
url_to_ips = {}
for line in open("access-log"):
        fields = line.split()
        ipaddr = fields[0]
        url = fields[6]
        # Create a set on first access
        if url not in url_to_ips:
           url_to_ips[url] = set()
        url_to_ips[url].add(ipaddr)
```

Again, we are faced with the problem of creating the first entry for each URL (hence, the check that makes the set on first access). We can't use Counter objects here, but not to worry—the defaultdict class is built just for this case. Here is an alternative implementation:

```
from collections import defaultdict
url_to_ips = defaultdict(set)
for line in open("access-log"):
        fields = line.split()
        ipaddr = fields[0]
        url = fields[6]
        url_to_ips[url].add(ipaddr)
```

After running this code, you could do things like find out which IP addresses are likely to be robots:

```
>>> url_to_ips['/robots.txt']
set(['173.11.97.115', '107.20.104.146', '61.135.249.76', ...])
>>>
```

defaultdict is a special Python dictionary that allows you to supply a callable for creating the initial entry to be used on first access. In the above code, we've specified that a `set` be used. Here are some examples to try:

```
>>> from collections import defaultdict
>>> a = defaultdict(set)
>>> a
defaultdict(<type 'set'>, {})
>>> a['x'].add(2)
```

```
>>> a['y'].add(3)
>>> a['x'].add(4)
>>> a
defaultdict(<type 'set'>, {'y': set([3]), 'x': set([2, 4])})
>>>
```

In effect, the function provided to defaultdict is triggered to create the first value whenever a non-existent key is accessed. Here are more examples:

```
>>> a['q']
set()
>>> a['r']
set()
>>> a
defaultdict(<type 'set'>, {'y': set([3]), 'x': set([2, 4]), 'r': set([]), 'q':
set([])})
>>>
```

Notice how entries for 'q' and 'r' were added simply by being referenced.

Underneath the covers, defaultdict uses a little-known special method called __missing__(). It's called on a dictionary whenever you read from a missing key. For example:

```
>>> class mydict(dict):
...     def __missing__(self, key):
...     return 0   # Return the missing value
...
>>> d = mydict()
>>> d['x']
0
>>> d['y']
0
>>>
```

Counter objects are implemented using the __missing__() function shown above. defaultdict objects create the missing value using a user-supplied function.

Dictionaries, Views, and Sets

One of the more subtle improvements to Python over the years has been related to the relationship between dictionaries and sets. In many respects, a set is just a collection of dictionary keys with no values. In fact, the underlying implementation of sets and dictionaries is very similar and shares much of the same code.

Despite their similarities, dictionaries have not traditionally provided a natural way to interact with sets of keys or values. Instead, there are simple methods to return the keys, values, and items as a list:

```
>>> a = { 'x' : 2, 'y' : 3, 'z': 4 }
>>> a.keys()
['y', 'x', 'z']
>>> a.values()
[3, 2, 4]
>>> a.items()
[('y', 3), ('x', 2), ('z', 4)]
>>>
```

Starting with Python 2.7, it is possible to express the keys and values of a dictionary as a "view" (which is also the default behavior of the above methods in Python 3). Unlike a list, a view offers a direct window inside the dictionary implementation. Changes to the underlying dictionary directly change the view:

```
>>> k = a.viewkeys()
>>> k
dict_keys(['y', 'x', 'z'])
>>> v = a.viewvalues()
>>> v
dict_values([3, 2, 4])

>>> # Now change the dictionary and observe how the views change
>>> a['w'] = 5
>>> k
dict_keys(['y', 'x', 'z', 'w'])
>>> v
dict_values([3, 2, 4, 5])
>>>
```

At first glance, it might not be immediately obvious how views are useful. On a superficial level, they support iteration, allowing them to be useful in many of the same ways as having a list. However, one of their unique features is the ability to interact with sets and other sequences more elegantly. To illustrate, here are some simple examples you can try:

```
>>> a = { 'x' : 1, 'y': 2, 'z' : 3 }
>>> b = { 'x' : 4, 'y': 2 }
>>> # Find all keys in common
>>> a.viewkeys() & b.viewkeys()
set(['y', 'x'])

>>> # Iterate over all keys except 'z'
>>> for k in a.viewkeys() - ['z']:
...     print("%s = %s" % (k, a[k]))
...
y = 2
x = 1

>>> # Make a set of all key/value pairs
>>> a.viewitems() | b.viewitems()
set([('z', 3), ('y', 2), ('x', 4), ('x', 1)])
>>>
```

In more practical terms, understanding the nature of views can simplify your code. For example, if you wanted to find all of the IP addresses that accessed your site but didn't look at the robots.txt file, you could simply write this:

```
>>> nonrobots = hits_by_ipaddr.viewkeys() - url_to_ips['/robots.txt']
>>>
```

Other Goodies: Queues, Ring Buffers, and Ordered Dictionaries

The collections module has a variety of other data structures that are also worth a look. For instance, if you ever need to build a queue, use the deque object. A deque is like a list except that it's optimized for insertion and deletion operations on both

ends; in contrast, a list has O(n) performance for operations that insert or delete items from the front of the list:

```
>>> from collections import deque
>>> q = deque()
>>> q.appendleft(1)
>>> q.appendleft(2)
>>> q
deque([2, 1])
>>> q.append(3)
>>> q
deque([2, 1, 3])
>>> q.pop()
3
>>> q.popleft()
2
>>>
```

If you specify a maximum size, a deque turns into a ring-buffer or circular queue:

```
>>> q = deque(maxlen=3)
>>> q.extend([1,2,3])
>>> q
deque([1, 2, 3], maxlen=3)
>>> q.append(4)
>>> q
deque([2, 3, 4], maxlen=3)
>>> q.append(5)
>>> q
deque([3, 4, 5], maxlen=3)
>>>
```

Last, but not least, there is an OrderedDict class. This is used if you want to store information in a dictionary while preserving its insertion order. This can be useful if you're reading data that you later want to output in the same order in which it was read. For example, suppose you had a file of parameters like this:

```
FILENAME foo.txt
DIRNAME  /users/beazley
MODE    a
```

You could read it into an OrderedDict like this:

```
>>> from collections import OrderedDict
>>> parms = OrderedDict()
>>> for line in open("parms.txt"):
...     name,value = line.split()
...     parms[name] = value
...
>>> p['DIRNAME'] '
/users/beazley'
>>> for p in parms.items():
...     print(p)
... ('FILENAME', 'foo.txt')
```

```
('DIRNAME', '/users/beazley')
('MODE', 'a')
>>>
```

Carefully observe how iterating over the dictionary contents preserves data in the same order as read.

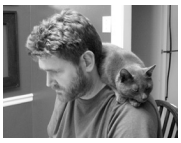
Final Words

If you're using Python to manipulate data, the collections module is definitely worth a look. Even if you've been using Python for a while, the contents of this module have been expanded with each new Python release. In modern Python releases, you might be surprised at what you find.

iVoyeur

Changing the Game, Part 4

DAVE JOSEPHSEN



Dave Josephsen is the author of *Building a Monitoring Infrastructure with Nagios* (Prentice Hall PTR, 2007) and is senior systems engineer at DBG, Inc., where he maintains a gaggle of geographically dispersed server farms. He won LISA '04's Best Paper award for his co-authored work on spam mitigation, and he donates his spare time to the SourceMage GNU Linux Project.
dave-usenix@skeptech.org

It started with the Vacation. It was only three or four days long, but vacations can be a dangerous time for me. They give me an occasion to pursue subject matter I might normally avoid for fear of the rabbit holes [1]. I'm sure you can relate. One poorly chosen Wikipedia article and a little down-time and suddenly it's 4 a.m. and you find yourself making cheese in the back yard, or thermite in the bathtub. Or was it cheese in the bathtub...? Well you get the point.

Anyway, a bit of heavy reading has lately left me with the palatable sense that my grasp of English grammar is not what it ought to be. That, in fact, it sucks. It's not a happy realization in a person who is paid money to write things.

Realization isn't the right word. I've been aware for some time that my West Coast public school education has left me deficient in this, and many other respects. To be sure, I have the innate grasp of grammar that we all share, but I've never, for example, diagrammed a sentence. Nor have I ever been instructed by a teacher to use the verb to find the subject of a sentence. Before yesterday I was wholly ignorant of how many types of verbs there are (transitive, irregular, dynamic, etc.). Seeing the skill with which these other writers put together words to make sentences to form thoughts, and comparing those sentences and thoughts to my own, has instilled in me a fascination with the rules of language syntax, rules which, if I knew them, would enable me to more accurately and completely (and let us all hope, tersely) articulate my thoughts. Have I been writing in the literary equivalent of Visual Basic my whole life? This is unacceptable.

Now, you and I, being the sort of people we are, the sort of people with training and experience in finding and consuming exactly the right knowledge—not just finding it and consuming it, in fact, but delighting in the finding and consuming of it—we have a penchant for cutting to the heart of things when we put in our cross-hairs subjects like English grammar. You and I aren't surprised to learn that the absolute best way to *understand* English grammar is to learn Latin, and being the sort of people we are, we're comfortable with that in the same way that we're comfortable with the knowledge that the best way to understand Perl is to know assembler.

To those around us, however—those outside the confines of whatever convention hall we happen to be occupying (and perhaps even those to whom we are married)—the idea of learning Latin is a strange, extreme, and probably elitist proposition. That you and I would even consider such an undertaking makes us, transitively, strange, extreme, and probably elitist people. I know this, even if I don't understand it, and so I maintain a cognitive duality to protect the anti-intel-

lectual sensibilities of my fellow man (and spouse). I tell myself I'm not learning Latin, that, in fact, I'm only learning *about* learning Latin, but I hesitate to mention even that to anyone off the conference floor, because that's just the sort of distinction a weirdo extremist would make. No, this undertaking must be a secret. We'll have to keep it between us.

It's vexing therefore, when I've fallen far enough into a rabbit hole that I find myself immersed in the study of "Latin for Mountain Men" [2] in secret, as if it were some kind of weirdo extremist samizdat/porn, to walk into the break room and hear someone announce:

"I have discovered the secret of speed eating. The secret is to make your meal broth heavy!"

Any sort of loudly asserted absurdity like this really shakes me up when I've been on a bit of a mental binge. It makes me feel somehow dissonant and inhuman. I've often suspected that these are the sorts of situations that make people like you and me become people like the Unabomber, so some time ago I developed a mental model to protect my psyche in these sorts of situations. I call it the "Inverse Feynman Filter". I'll let Dick explain:

> I had a scheme . . . when somebody is explaining something that I'm trying to understand: I keep making up examples. For instance, the mathematicians would come in with a terrific theorem, and they're all excited. As they're telling me the conditions of the theorem, I construct something which fits all the conditions. You know, you have a set (one ball)—disjoint (two balls). Then the balls turn colors, grow hairs, or whatever, in my head as they put more conditions on. Finally they state the theorem, which is some dumb thing about the ball which isn't true for my hairy green ball thing, so I say, 'False!' [3]

If a Feynman filter is a mental model for the simplification of complex theorems, my Inverse Feynman Filter is a mental model for the complication of the absurd and idiotic. For example, this speed eating of broth thing sounds to me like a data compression or maybe a signal processing problem. You're taking food, and compressing it to broth, so it can be processed more quickly. See how wonderfully that works? If the subject matter is data compression, we needn't concern ourselves with why someone would want to speed eat, much less that someone felt the necessity to contemplate its secrets. We can ignore entirely the question of whether "heavy" is modifying "meal" or "broth" (why would the weight of the meal-broth matter?) and his improper use of "of" (assuming he meant that he'd discovered the secret *to* speed eating), so bonus, our top secret grammatical endeavors remain undiscovered! We can even interact, observing, for example, that some types of data are more difficult to compress, like so:

"That's fascinating. How exactly does one 'make' one's meal 'broth heavy'? How would I, for example, go about 'speed eating' sunflower seeds?"

"No, no. If you want to speed eat, you have to eat broth."

"So your 'discovery' is really that you can drink soup quickly?"

"Well, yeah, I guess."

As you can see, it's not a perfect model. When it fails, I simply transition to my backup technique, which is to use the story as a lengthy intro to a monitoring column for *;login:*. This way I can focus my mental energy toward the creation of

an appropriate segue from the story into the proper subject matter of the column. The speed eating of broth is, for example, a perfect metaphor for Mathias Kettner's Nagios plugin, Check_MK [4].

Centralized polling engines like Nagios are difficult to scale because the load increase is linear. Every new service on every new host makes the monitoring system work that much harder. Eventually, you'll hit an upper-level limit on the number of services a single poller can handle. Depending on the polling interval—whether you're processing performance data locally, and the sheer horsepower of the hardware on which the poller is running—that limit is generally around 5000 services. At that point we need to look at splitting the workload across multiple pollers. Various means exist to split and parallelize the polling workload, most of which I've described in this column at one time or another.

But what if we could "brothify" some of those service checks, so that instead of performing seven service checks on a host, we could perform a single check on the host that would return the status of all seven services? This isn't a new idea—there have been various attempts to brothify and speed eat service checks over the years—but the idea hasn't caught on, because the implementations were problematic. To be fair, the problem is really the design of Nagios, which assumes that every check returns a singular result from an individual service. The configuration associated with the brothification of multiple service checks is therefore invariably some kludgey mess involving passive service checks which, not unlike brothifying sunflower seeds, is just not worth the effort.

When one considers the inevitable differences between various types of hosts, that some will run services others won't, and that some will use alert thresholds that are more or less strict, it's easy to imagine the configuration nightmare associated with our broth. The specifics of what to monitor and how to monitor it will either need to be moved off the poller and out to the monitored hosts, abdicating the advantages of centralized configuration, or into the check command itself such that the check command for each host is accompanied by three pages of options, only a few of which are actually specific to that host.

Check_MK solves all of these problems and more, providing not just a means to brothify all the service checks on a host, but an all-inclusive monitoring agent that dynamically detects and reports a litany of information about the host. Perhaps "solves" is a strong word, as the server-side configuration is still a mess involving passive service checks, but Check_MK creates and manages all of that for you. In a way, the plugin's dynamic configuration is the most impressive thing about it. After installing the plugin server side, and the agent on the host, the Admin runs an inventory program, which dynamically detects and, through the clever use of Nagios templates, generates the complete server-side configuration for every host inventoried, including the active check for the host as well as the passive checks for each service detected. This was a heavy lift; the kind of programming few of us enjoy.

The agent is tiny, being a shell script running under xinetd on Linux. Unlike NRPE [5], no attributes or arguments are passed from the server, which limits the vulnerability footprint. The agent is easily extended for custom broth ingredients by way of a plugin directory into which the admin may drop his own scripts. These custom scripts will be called by the MK agent, and, assuming they follow some simple formatting rules, their output will be parsed by the server plugin without any addi-

tional configuration. The plugin will in turn generate passive checks for them and report them back to Nagios.

By default the agent dispenses a broth with the big four food groups—CPU, RAM, disk, and network—auto-detecting in the process CPU numbers, NICs (including virtual interfaces like tun/tap) and even disk partitions. A dizzying array of other data is thrown in for flavor, including a process list, and a host of hardware-specific info about devices like NVIDIA and 3-Ware cards, ACPI, and on and on. An agent is even available for Windows which includes all sorts of Windows and Active Directory metrics. A full list can be had by calling the plugin on the command line with a -M switch.

The agent program passes status to the plugin in a way that draws a distinction between mere service state and performance data. The plugin is, in turn, aware of performance data, which it can send to an RRDtool front-end for Nagios called "PNP4Nagios" [6]. The plugin even automatically generates the appropriate action_url syntax in the Nagios configuration so that the performance data graphs generated by PNP4Nagios are displayed on the Nagios Web Interface, all without the admin needing to lift a finger.

The Check_MK plugin provides hooks to customize the configuration it generates, making it easy to specify alert thresholds for individual services on individual hosts. The rules are implemented as a cascading series of defaults, with the most specific match winning. It can also query SNMP devices such as routers and switches using snmpwalk in lieu of a host-side agent.

It's possible that by mixing our service checks together into a broth, we might learn something about how they interact. The Check_MK plugin has a few neat features that explore this possibility, including the ability to detect the primary node in an HA-Cluster using service information returned by the agents, and a feature called "Service aggregations." This latter is an attempt to capture business logic, and bears some explanation because it's actually quite a powerful idea.

A service aggregation can be thought of as a virtual service that is made up of several real services: for example, one can imagine a virtual service called "Email," which is made up of the qmail-send daemon on several hosts along with a few database and HTTP processes on various other hosts throughout the infrastructure. If any of these individual services goes down, Check_MK marks the top-level virtual service down as well.

Check_MK is a well-designed system that should be considered as a replacement for NRPE, especially if you're experiencing growing pains. Next time I'll be covering another Mathias Kettner creation called "MK_Livestatus," which is actually an idea he stole from me about a year before I had it. Until then, look out for the rabbit holes.

Take it easy.

References

[1] "Down the rabbit hole": a metaphor for adventure into the unknown, from its use in *Alice's Adventures in Wonderland*.

[2] Latin for Mountain Men: http://mysite.du.edu/~etuttle/classics/latin/learnlat.htm.

[3] *Surely You're Joking Mr Feynman*, p. 85: http://books.google.com/books?id=7papZR4oVssC&pg=PA85&lpg=PA85.

[4] Check_MK home page: http://mathias-kettner.de/check_mk.html.

[5] NRPE, Nagios Remote Plugin Executor: http://exchange.nagios.org/directory/Addons/Monitoring-Agents/NRPE--2D-Nagios-Remote-Plugin-Executor/details.

[6] PNP4Nagios: http://docs.pnp4nagios.org/pnp-0.6/start.

/dev/random

ROBERT G. FERRELL

Robert G. Ferrell is a fourth-generation Texan, literary techno-geek, and finalist for the 2011 Robert Benchley Society Humor Writing Award.
rgferrell@gmail.com

While perusing HyperDex—because anything with "hyper" in it draws my geekish attention like a Texas ice storm draws tow trucks—I saw that they use n-dimensional Euclidean mapping for storing and manipulating data. I happen to be enamored of the concept of and, verily, even the phrase n-dimensional, largely due to my (over)use of it in a couple of science fiction novels I wrote. Now, the inclusion of Euclidean sort of ruins it for mind-boggling potential, because Euclidean space is, in fact, composed of only three actual dimensions. Sure, that's an oversimplification. I like those. They are tidy and fit neatly on the page. I've made hundreds of them over the years. You can keep score at home, if you like.

Ignoring, for the time being, the Father of Geometry (and his college roommate, the Third Cousin Once Removed of Doodling in the Margins), let us turn our attention instead to a further examination of the relationship between extra dimensions and computing. The idea that data structures can be visualized in two or three dimensions is nothing new, of course: every beginning programming student learns about matrices. RAID configurations fall into a similar category, albeit more on the hardware side, in that data are striped across multiple disks and therefore exist conceptually in Euclidean space. All of this, while moderately interesting, is too mundane for my tastes. No, what I'm talking about is n-dimensional quantum computing. Booyah.

Let me break it down for you: quantum superposition is all well and good, but it's still limited by the fact that there are only three possible states for a qubit to possess: one, zero, and both. In my n-dimensional hyper-quantum ("ND-Hype") computer, each qubit can itself exist in infinite multiple dimensions at once, providing a hyper-exponential increase in computing power.

One of the hurdles of the current effort to build a useful quantum computer is generating enough entangled qubits to get anything useful done. The ND-Hype computer obviates that problem quite neatly in that it only requires one qubit to operate. When additional logic gates are needed to solve a problem, it dynamically recruits entangled brethren in as many dimensions as necessary, effectively giving it infinite qubits with which to work. Since the extradimensional qubits are all entangled with the original, their quantum states are transmitted to it instantaneously in what I will, naturally, call "hyperpositioning."

Thus, with but a single qubit, you can possess functionally limitless computing power. Designing the user interface might present a bit of difficulty, admittedly, since infinite simultaneous data outputs would be a little problematic to display,

but that's an issue for the GUI engineers. I suppose you'd need a fairly large amount of RAM and one heck of a data bus to hold all that information long enough to accomplish anything with it, too. Hey, I'm just the idea man here.

I think one of the reasons I'm drawn to all things quantum is that every aspect of that topic is utterly counterintuitive and bordering on the insane, and, well, like attracts like. I mean, think about it: in the quantum world, the answer to the age-old philosophical question, "If a tree falls in the forest and no one is there to hear it, does it make any sound?" is: "yes and no." Unless there is an observer, the outcome has both possible states, not one or the other. That is clearly insane.

The only way I can even begin to grasp it is to imagine that every single binary event I can witness, or even imagine, has a universe where the outcome is one, and another where it is zero. Once there is a witness to the precipitating event, the outcome is fixed and that observer is tied irrevocably to that universe. Until and unless there is an observer, however, the outcome is indeterminate. In my novels *Tangent* and *Infinite Loop* I refer to those quantum temporal inflection points as "frames." Whatever term you choose to refer to this phenomenon, "rational" does not enter into it.

But, back to HyperDex. I think their data storage and retrieval scheme is pretty slick, although it also smacks of Yet Another Insidious Cloud Computing Initiative. At least they track the precise location of your data in their n-dimensional space, which is far more than I can say for most true cloud applications. I think my next big think tank project, based on a dream I had the other night, will be to create a sort of air traffic control system for use in the cloud that allows a user to track where her data actually are at a given moment in both logical and physical space. Sort of like tracking your teenage son when he borrows your car to go the store for you via four friends' houses, the skateboard park, two different malls, and the Hyperdodecahedroplex Theater out on Route 15. And then comes home six hours later without the milk or bread. Not to mention an empty gas tank.

I'd have a huge map of the world for my data tracker like the one in *War Games* that would trace each piece of your data as it spreads out through the cloud, displaying it like radar tracking of missile launches. This would help to drive home the message that the cloud is a colossal global security risk while at the same time providing an impressive component for your next dog-and-pony affair. It might even lead to a game show called *Guess Where Your Data Will Go Next.*

"Guess where your data will go next. Hands on buzzers, ladies and gentleman. Go!"

Buzzz! "Oslo?"

Buzzz! "Seville?"

Buzzz! "Dubuque?"

"No, I'm sorry, contestants, the correct answer was Baku, Azerbaijan. Ed, tell our contestants what they get for being such lose...good sports."

"Certainly, Bob. All contestants on *Guess Where Your Data Will Go Next* receive a year's supply of pre-compromised firmware and a copy of our home game, *Guess Where That Thumb Drive You Found in the Parking Lot Has Been.*"

I should probably stop eating those habanero Brussels sprouts right before bed.

Book Reviews

MARK LAMOURINE, BRANDON CHING, PETER H. SALUS, JEFF BERG, EVAN TERAN, AND RIK FARROW

Jenkins: The Definitive Guide
John Ferguson Smart
O'Reilly Media, 2011. 406 pp.
ISBN 978-1-449-30535-2

Anyone who's worked on the development side of system administration for any length of time has probably put together some form of automated build system. Those of us who are older probably did it from scratch.

A corollary of the Agile development process is the need for a strong automated build and test service. The popularity of Agile development has led to the improvement of services to manage the build and test process and to provide visibility at each step. To oversimplify, the idea is to rebuild and retest with every code check-in. Ideally, little bugs are discovered early and fixed before they grow into big problems. Of course, people have given this a name: continuous integration. Jenkins is a fairly recent addition to the tool set.

It's important to understand at the outset that Jenkins doesn't really do anything by itself. Jenkins manages and coordinates a series of activities that would otherwise be done manually or not at all.

Jenkins: The Definitive Guide is light on philosophy. The introduction and justification for using continuous integration in general and Jenkins in particular takes just eight pages. That includes a three-page seven-step timeline to get from no automation to complete continuous integration. The rest of the book actually does a fine job of filling in the outline but doesn't waste any effort trying to proselytize.

The book is both sparse and dense. Jenkins integrates with dozens of other pieces of software, and the author doesn't try to hand-hold the reader through any of them. For example, Jenkins is a Java application. Java installation and verification take a single paragraph with a URL reference to the Oracle Web site for download and installation instructions. Likewise, Git installation and creation of a GitHub account to download the sample jobs for the next chapter take just over a page.

Each chapter covers a high-evel task for the user. These correspond to the configuration of different Jenkins activities. They also match the timeline presented in Chapter 1 for gradually introducing continuous integration.

The first four chapters cover the introduction, preparation of the environment, and installation and configuration of Jenkins. The example project for the book is hosted on GitHub and the book instructs the reader to create an account and fork the sample repository. I'm not sure how I feel about reference books which depend on live commercial services. Realistically, the book is likely to be obsolete before GitHub goes belly up, but it bothers me somehow.

The remaining chapters cover creating build jobs, automated testing and notification, authentication and access control mechanisms, automatic code quality scanning and reporting, and automated deployment. The final chapter talks about updates, configuration backups, and storage management and server loading. I was glad to see that because I didn't see anything about capacity planning at the beginning.

On top of the standard features there is a long and growing list of plugins for open source and commercial applications available, each of which will require some knowledge of the related software. I think this is a good thing, but it's clear that this book (and this software) is not for the novice developer or sysadmin.

Jenkins was written in Java and to manage Java projects. This shows in the focus of the examples in the book. The plugin list, though, shows that Jenkins is being used to manage projects in a dizzying array of environments. All of the current major scripting languages and Web frameworks are represented, as are a number of more obscure (to me) tools. The book doesn't cover these, but if a plugin exists for your language or environment it doesn't look like it will be difficult to install or configure.

Jenkins is managed almost entirely through the Web user interface. The book is full of screen shots illustrating the text. One concern is that any changes to the visual layout of the user interface will make portions of the book obsolete.

User interface evolution isn't a problem limited to books and software with graphical interfaces, but I think it could be a greater problem for them.

"Guide" is an appropriate term to use for this book if you take it in the sense of a tour rather than a reference. Each chapter shows you something important and concrete but then leaves you at the entrance to some new place for you to explore on your own, since each subject is covered well elsewhere. Take the tour and then decide what parts you want to explore next and in greater depth.

This book is licensed by the author under the Creative Commons license. The content is available on the Internet at DocBook. This is a case of Tim O'Reilly putting his money where his mouth is. He's spoken out publicly against draconian government-enforced content monopolies. He's willing to publish CC licensed books. He knows how to decide what will make money, and he's convinced that he doesn't need an exclusive copyright to publish a book worth buying. And he's right.

—Mark Lamourine

Webbots, Spiders, and Screen Scrapers: A Guide to Developing Internet Agents with PHP/cURL, 2nd Edition

Michael Schrenk

No Starch Press, 2012. 362 pp.
ISBN 978-1-59327-397-2

In the second edition of *Webbots, Spiders, and Screen Scrapers: A Guide to Developing Internet Agents with PHP/cURL*, Michael Schrenk introduces you to the world of automated webbots and scripts that can filter, parse, store, and process Web-based information that suites your needs. Using the massive information available online and through the methods described in this book, you can wield that data almost any way you want. Need to collect metrics or parse and store content for your academic research? What about wanting to have an email sent to you when your bank account gets low? Perhaps you are watching the price on an eBay auction and you want to be notified—or even have your script automatically bid for you—when it hits a certain amount. Webbots can do all of these things and more.

The book is broken up into four parts comprising 31 chapters. While this sounds like a lot, most chapters are short and to the point, covering a specific topic and guiding you to external sources where appropriate. The first section covers foundational techniques and technologies and introduces you to the PHP language and the cURL library. The second section delves into some simple projects of common Web automation tasks and demonstrates the general script flow and how specific libraries function to get results. Some example projects here are price monitoring, form submission, aggregation, and email-reading/sending webbots. Section three covers advanced topics such as dealing with cookies, encryption, authentication, and macros. Finally, section four covers general webbot considerations such as fault tolerance, stealth, and webbot-friendly Web design.

As someone who has been writing PHP/cURL webbots and spiders myself for about seven years now, I was particularly excited to read this book. I think Michael does a good job of covering most of the basic techniques and challenges having to do with webbot scripting. Although there are a number of good chapters that will get you up and running quickly, I do feel that most of the chapters were a bit cursory, lacking some very important coverage of alternate tools and techniques. Also, Michael doesn't delve deeply into topics such as AJAX and complex JavaScript-driven sites and forms. In my experience, a large number of sites use these kinds of techniques, and without the right tools they can be very difficult to parse.

Overall, *Webbots, Spiders, and Screen Scrapers* is well-written, easy to follow, and will get you started quickly. Having said that, its lack of depth in certain areas definitely makes it most appropriate for beginning developers/scripters.

—Brandon Ching

A Culture of Innovation: Insider Accounts of Computing and Life at BBN

David Walden and Raymond Nickerson, eds.

Waterside Publishing, 2011. 559 pp.
ISBN 978-0-9789737-0-4

We owe a great deal to the concept of the industrial lab, the first of which was that of Thomas Edison in Menlo Park, NJ (now renamed Edison). World War II gave rise to IBM's Research Division in Manhattan and, since 1970, in Yorktown Heights, NY. It also saw the start of Bolt Beranek and Newman in Cambridge, MA, in 1948.

This volume incorporates the narratives of 19 of those who were involved in BBN over a period of 60 years. It is not 100% new material. Some of the contents originally appeared in two special issues of *IEEE Annals of the History of Computing* (v. 27.2 and v. 28.1 [2005, 2006]), but many of those articles reappearing in this volume are expanded versions, and some of the chapters are completely original to this work.

Dick Bolt and Leo Beranek founded their partnership in 1948. Their first job was the acoustical engineering of the not-yet-built UN headquarters in New York. Bob Newman became

a partner and the name was changed in 1950. I am not going to discuss acoustics, but the Harvard Electro-Acoustics Lab and the MIT Psycho-Acoustic Lab underlie all the work of the following decades. BBN grew rapidly. By 1960, there were offices in Los Angeles and Chicago; there were 128 employees in Cambridge, 22 in LA, and three in Chicago.

One of the "bright young men" at MIT was J.C.R. Licklider. In 1962, Lick went to Washington, to ARPA, where he became a prime mover in the expansion of computing and in what would become the ARPANET. In 1968 BBN's response to the RFP for the ARPANET was complete. In October 1969 the first two IMPs (Interface Message Processors), one at UCLA, one at the SRI in Menlo Park, CA, communicated with each other: the 21-year-old corporation would make key contributions to a technology that was to change the world.

But the ARPANET/Internet was far from the end of BBN's innovations.

BBN had been the "experimental" site for DEC's PDP-1; thanks to Licklider, Dick Pew, and their cohort, the field of human-computer interaction came into being; networked email was born here (thanks to Ray Tomlinson; pay no attention to the Shiva Ayyadurai silliness); time-sharing was demonstrated here, etc.

I don't want to turn what should be a review into a menu, but if you are reading *;login:*, you owe a major debt to BBN: acoustic signal processing, control systems, torpedo data analysis, several medical applications, educational technology, speech processing, and natural language understanding are merely a few of the topics discussed in this volume.

Not everything is an easy read; some chapters are better than others. But the work is quite significant in that you can hear the voices of a number of remarkable individuals. I learned and profited from every chapter.

One of the most fascinating is Stephen Levy's "History of Technology Transfer at BBN," covering 1948–1997 (the last decade is covered in Walden's "Epilog"). I had not realized the number of business relations of various kinds BBN (and successors) had entered into, nor what a large percentage of the exchange of agreements was profitable.

Although currently a branch of Raytheon, BBN still functions as a research and development facility, unlike Bell Labs or XeroxPARC. This volume is a fitting monument. If you are at all interested in technological history, *A Culture of Innovation* is more than merely a worthwhile investment. All the contributors deserve my thanks; Walden and Nickerson, my gratitude.

—Peter H. Salus

The Tangled Web: A Guide to Securing Modern Web Applications

Michal Zalewski

No Starch Press, 2012. 268 pp.
ISBN 978-1-59327-388-0

Michal Zalewski succeeds in condensing into a single comprehensive volume topics that could easily fill several books, and he provides the right reader with exactly what he or she needs to know regarding the Web, the browsers we use to navigate it, and the considerations we need to be aware of to secure it. Depending on your experience with these topics, this is not a breeze-through read; this book requires time and attention to grasp and hold the topics covered.

The book is split into three parts: "Anatomy of the Web," "Browser Security Features," and "A Glimpse of Things to Come." The first third of the book really lays the groundwork for the remaining portions, giving the reader the necessary background to understand what makes the browsing experience work. Zalewski covers the basics, devoting entire chapters to the makeup of a URL, the Hypertext Transfer Protocol, HTML, Cascading Style Sheets, browser-side scripts, non-HTML document types, and browser plugins. As I'll expand on later in this review, Zalewski crams enormous amounts of information into the 15–20 pages (on average) per chapter. Whether it is technical detail or history behind the design of a browser or standard, he leaves no stone unturned. The second third delves specifically into security features of browsers, languages, and plugins, with chapters on topics such as content isolation logic, origin inheritance, content recognition mechanisms, and handling rogue scripts. The last third of the book looks at security features and browser mechanisms that are expected to emerge. A chapter is devoted to common Web vulnerabilities.

The level of detail Zalewski goes into is excellent. Topics such as browser history, design considerations and behavior, HTML markup, security features in plugins, and secure coding are covered with minimal filler and with examples that illustrate discussions throughout. Only rarely is the reader left with questions, and the text provides detailed references pointing the reader toward additional information if necessary. As an example of the depth that Zalewski provides for a given topic: the HTML chapter covers everything from a discussion of the RFC and subsequent HTML version evolution to browser parsing behavior in handling different code segments, including the role UTF-8 characters can play in manipulating a browser's parsing behavior. Zalewski acknowledges that the book is not all-encompassing in certain areas; however, I would argue that it's as close to comprehensive as is necessary. For those who get spun around in

the details, there are "Engineering Cheat Sheets" at the end of each chapter summarizing major points made throughout.

I would recommend this book without question for any Web application developer. The information within is essential knowledge to be applied in everyday efforts. I'd also pose it as essential reading for security professionals—researchers, analysts, penetration testers, etc.—who will touch the Web application space. As a member of this community, I can say the information presented is just another useful tool in the old shed that will be applicable at some point. Given the detail, you may want to keep it around to thumb through on the fly.

—Jeff Berg

A Bug Hunter's Diary: A Guided Tour Through the Wilds of Software Security

Tobias Klein

No Starch Press, 2011. 208 pp.
ISBN 978-1593273859

A Bug Hunter's Diary is a unique book. Its approach to discussing the topic of computer security is completely different from any other I've read, and that's a good thing. Instead of the usual "this is what could be done," this book says "this is what I did and why."

What makes this book so different really boils down to two things:

- The level of detail given when discussing the bugs is extremely high. You will need a working knowledge of C or C++, and assembly (usually x86) wouldn't hurt either.
- The format of the book is literally that of a diary, which makes it more of a unique read.

There are eight chapters—an introduction followed by in-depth analysis of seven major bugs that the author found and developed successful exploits for.

The introduction is a good overview of the different approaches that are applicable to this type of work, ranging from static analysis to runtime analysis with a debugger to fuzzing. The author very much prefers static analysis but is quick to point out that each approach has its pros and cons and that everyone will have their preference.

Each "diary entry" is broken down into the steps that the author took to develop the exploit and closes each one with two very useful things: "vulnerability remediation," which discusses what the vendors did to patch the problem and how long it took, and "lessons learned," usually a short list basically describing some rules of thumb by which the problem could have been avoided entirely.

So what types of bugs are we talking about here? Pretty much exclusively ones that fall into the category of memory corruption. While things like XSS and other higher-level bugs are very popular now (and are just as serious), they aren't addressed in this book. These are "hard core," low-level bugs which require an intimate knowledge of the languages, operating systems, and architectures being exploited. For those not very familiar with these types of memory corruption bugs, I highly recommend reading the article "Smashing the Stack for Fun and Profit." It's an older *Phrack* article, but really spelled out the basics of how memory corruption bugs can be leveraged by an attacker.

Of course the book isn't perfect. The level of detail, while extremely impressive (and useful), can be a bit overwhelming. It's not that there is too much information to take in; it's more about presentation. Just about every page is half filled with code listings of some kind, which is great but also can make the flow of reading a little difficult. You will probably find yourself jumping back and forth between the code listings and the descriptions several times before you have that "ah-ha" moment and see what the bug is. Fortunately, the author puts the most relevant lines of code in bold so at least you know what you are supposed to be looking at.

All in all, this is a great book, especially for those who have a strong background in C or C++ programming and want to learn how to think like a security engineer.

—Evan Teran

D is for Digital: What a Well-Informed Person Should Know About Computers and Communications

Brian Kernighan

DisforDigital.net, 2011. 223 pp.
ISBN 978-1463733896

Sporting a white cover with blue lettering, *D is for Digital* mimics the look of classic Kernighan books. But the target audience for this book is not programmers, but, rather, educated people who are not CS majors.

Brian writes in his foreword that he has been teaching a class at Princeton called "Computers in Our World" since 1999, and his experiences teaching what people need to understand about computers for over a decade really shows in his book. *D* is not a textbook, but a gentle and clear journey that covers hardware, software, and communications in 12 chapters. I kept picking the book back up and reading more, partially for

the history embedded in it and partially because I enjoyed learning just how Brian approaches difficult topics. I had toyed with writing a book about computers a long time ago, but got bogged down in my explanation of binary. Brian has no problem with covering binary, assembler, file systems, JavaScript, Web bugs, and traceroute, while keeping the tone light and readable.

D is split into three sections: hardware, software, and communications. Communications covers the Internet, but also some communication hardware, cryptography, security, and privacy issues. If this seems like a lot to cover in just over 200 pages, the goal is not to overwhelm the reader, but to provide a solid background. The chapters on the Web and privacy are worth the price all by themselves.

There are no footnotes or references, in keeping with the style of the book. There is a list of resources at the back and a glossary. There is also an index, so if the reader knows what she is looking for, ADSL or "bit rot," she can find a good explanation for it in this book.

D makes a great gift for the person who is always asking questions, or perhaps for someone who really needs to know what he or she is talking about. I wish that this book were required reading for anyone attempting to write legislation related to computers, the Internet, and online privacy. I did find myself wondering what level of education should be expected of the target audience, and settled for anyone who has at least two years of college. Also, the book can be used as a reference, in that any part of the book can be read in isolation.

—Rik Farrow

NOTES

USENIX Member Benefits

Members of the USENIX Association receive the following benefits:

Free subscription to *;login:*, the Association's magazine, published six times a year, featuring technical articles, system administration articles, tips and techniques, practical columns on such topics as security, Perl, networks, and operating systems, book reviews, and reports of sessions at USENIX conferences.

Access to *;login:* online from October 1997 to this month: www.usenix.org/publications/login/

Discounts on registration fees for all USENIX conferences.

Special discounts on a variety of products, books, software, and periodicals: www.usenix.org/membership/specialdisc.html.

Contributing to USENIX Good Works projects such as open access for papers, videos, and podcasts; student grants and scholarships; USACO; awards recognizing achievement in our community; and others: http://www.usenix.org/about/goodworks.html

The right to vote on matters affecting the Association, its bylaws, and election of its directors and officers.

For more information regarding membership or benefits, please see www.usenix.org/membership/ or contact office@usenix.org, 510-528-8649.

USENIX Board of Directors

Communicate directly with the USENIX Board of Directors by writing to board@usenix.org.

PRESIDENT
Clem Cole, *Intel*
clem@usenix.org

VICE PRESIDENT
Margo Seltzer, *Harvard School of Engineering and Applied Sciences*
margo@usenix.org

SECRETARY
Alva Couch, *Tufts University*
alva@usenix.org

TREASURER
Brian Noble, *University of Michigan*
noble@usenix.org

DIRECTORS
John Arrasjid, *VMware*
johna@usenix.org

David Blank-Edelman, *Northeastern University*
dnb@usenix.org

Matt Blaze, *University of Pennsylvania*
matt@usenix.org

Niels Provos, *Google*
niels@usenix.org

CO-EXECUTIVE DIRECTORS
Anne Dickison and Casey Henderson
execdir@usenix.org

USENIX Announces New Co-Executive Directors

Margo Seltzer, Vice President and President-elect, USENIX Board of Directors

Greetings, USENIX Members. I am delighted to let you know that the USENIX board has concluded its search for a new Executive Director.

One of the best signs of a healthy organization is when you can hire from within, and it is with great pleasure that I present Anne Dickison and Casey Henderson as the new USENIX Co-Executive Directors. Anne and Casey have been with the Association for 8.5 and 9.5 years, respectively. Anne has been our Marketing Director, while Casey has been our Information Systems Director. As Co-Executive Directors, they will continue to provide vision, oversight, and direction in their respective functional areas, as well as sharing the Executive Director's responsibilities. Each of them has shown initiative in her job, creativity, and an ability to get things done, and has been instrumental in helping USENIX through the transition we've undergone in the past six months. Their complementary skill sets offer USENIX a unique opportunity to simultaneously ensure continuity and a smooth transition while setting us up for new initiatives. The Board and I are 100% confident in their ability to lead the organization into the future.

Anne and Casey have assumed the position of Executive Director effective April 2, 2012. Mail to execdir@usenix.org will

Casey Henderson and Anne Dickison, USENIX Co-Executive Directors

be directed to them, and that is the address you should use to contact USENIX, when you aren't looking for a particular person or function.

It has been my pleasure to have had the opportunity to serve USENIX as its Executive Director for the past six months. I have enjoyed working closely with Anne and Casey and the rest of the USENIX staff. They are all true professionals, who are deeply committed to the Association and its goals. I hope you will join me in congratulating Anne and Casey on their new roles and will continue to work with them as you have always done.

Notice of Annual Meeting

The USENIX Association's Annual Meeting with the membership and the Board of Directors will be held on Wednesday, June 13, 2012, in Boston, MA, during USENIX Federated Conferences Week, June 12–15, 2012.

Results of the Election for the USENIX Board of Directors, 2012–2014

The newly elected Board will take office at the end of the Board meeting in June 2012.

PRESIDENT
Margo Seltzer, *Harvard School of Engineering and Applied Sciences*

VICE PRESIDENT
John Arrasjid, *VMware*

SECRETARY
Carolyn Rowland, *National Institute of Standards and Technology (NIST)*

TREASURER
Brian Noble, *University of Michigan*

DIRECTORS
David N. Blank-Edelman, *Northeastern University*

Alexandra (Sasha) Fedorova, *Simon Fraser University*

Niels Provos, *Google*

Dan Wallach, *Rice University*

Conference Reports

In this issue:

10th USENIX Conference on File and Storage Technologies (FAST '12)

San Jose, CA
February 14–17, 2012

Opening Remarks and Best Paper Awards

Summarized by Rik Farrow (rik@usenix.org)

Bill Bolosky began the conference with the statistics. FAST 2012 set lots of records: largest number of submissions and accepted papers, lowest acceptance rate, and largest number of attendees. Bill wondered if it was just that the economy is getting better, or that the conference is that popular.

They also tried something new this year: short papers that are refereed the same way as longer papers. Bill then showed an image representing the words found in paper subjects and abstracts. Obvious words, such as "storage" and "system," were most prominent, followed by "file," "data," "deduplication," "flash," and "performance" (approximately—check out the video). "Cloud" is still a tiny word, but Bill expects that will grow.

Jason Flinn took over and described the first Test of Time award, for ideas that appeared at FAST over 10 years ago; it was presented to Sean Quinlan and Sean Dorward for "Venti: A New Approach to Archival Storage." Next, he announced the Best Paper awards: "Recon: Verifying FS Consistency at Runtime," by Daniel Fryer et al., and "Revisiting Storage for Smartphones," by Hyojun Kim et al.

Keith Smith of NetApp and Yuanyuan (YY) Zhou of UCSD will be the chairs of FAST '12.

Implications of New Storage Technology

Summarized by Dutch Meyer (dmeyer@cs.ubc.ca)

De-indirection for Flash-based SSDs with Nameless Writes

Yiying Zhang, Leo Prasath Arulraj, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau, University of Wisconsin—Madison

Yiying Zhang presented her research on a new "Nameless Writes" interface to solid state disks (SSDs). SSDs generally include a Flash Translation Layer (FTL) that maps logical to physical addresses. This allows the device to perform wear-leveling behind an opaque address space. However, this indirection incurs mapping table space cost and performance overheads. Zhang proposed eliminating these overheads by largely removing the address translation table and storing physical data addresses directly in the file system. In the proposed interface, file systems do not specify a logical address when issuing a write. Instead, the SSD acknowledges completed write requests with the data's physical address.

Realizing a nameless writes interface required addressing a number of problems. Writes are always placed in new physical locations, which forces metadata modifications to cascade address updates to new physical references up the length of the file system tree. To address this problem, Zhang uses a traditional write interface for file system metadata, including traditional on-device address translation. Since file systems generally store far more data than metadata, the cascading updates can be eliminated while preserving most performance and cost advantages. Physical address migration is also a challenge. With nameless writes, SSDs move data beneath running file systems, just as they do today. This requires that callbacks be sent to the file system informing it of any planned moves, while a temporary mapping table in the SSD ensures that requests are routed appropriately.

To evaluate their system, Zhang and her team created an SSD emulator, ported ext3 to the nameless writes interface, and evaluated against a page-mapped FTL, a hybrid FTL, and a nameless-writes FTL. The effort required 4360 lines of code changes to ext3. Their SSD emulator operates as a pseudo-block device and stores results in memory. They found that their nameless writes indirection mapping table required only 2%–7% of the metadata overheads and performed up to 20 times better on a workload of random writes.

Following the talk, Geoff Kuenning from Harvey Mudd College asked the community why we don't simply write a file system that is designed specifically for the SSDs, and remove the FTL entirely. Zhang replied that she thought the device should control its own wear leveling. Margo Seltzer asked if the space savings from removing the FTL mapping table is offset by the temporary mapping tables. Zhang explained that the overall metadata requirements are much smaller. Keith Smith from NetApp asked whether the authors had considered a richer interface, an idea that Zhang thought was promising. Ethan Miller from UCSC followed up on Smith's question to note that an object interface would provide the same benefits, even though it moves much of the management from the file system to the device.

The Bleak Future of NAND Flash Memory

Laura M. Grupp, University of California, San Diego; John D. Davis, Microsoft Research, Mountain View; Steven Swanson, University of California, San Diego

Laura Grupp presented her team's ominously titled paper, "The Bleak Future of NAND Flash Memory." Their goal is to project the evolution of NAND-based flash into the year 2024 to determine if the current reliability and performance will be derailed by technical limitations. Their findings are mixed. By some metrics, flash will continue to improve, but in other ways it will decline.

It is widely understood that current NAND-based flash drives are fast and reliable but have a relatively high cost-to-capacity ratio. Moving forward, capacity will no doubt increase, but with current processes and technologies, the increased density will incur increased error rates and decreased performance. To anticipate the future of NAND flash, Grupp and her team combined measurements of modern flash architectures with projected trends in manufacturing to model the capacity, latency, and throughput of flash going into the future.

Grupp explained that capacity will increase with the bit density of each cell. Current technologies include single-level cells (SLCs), which store a single bit; multi-level cells (MLCs), which store two bits; and triple-level cells (TLCs) which store three bits but are not really triple-level cells (they have eight levels). Cell size decreases by scaling, following Moore's Law. Current processes are between 25 nm and 34 nm, with industrial working groups predicting 6.5 nm by 2024. These factors suggest a 43-fold increase in capacity over that period. To test performance, Grupp and her co-authors used an in-house testing system to analyze 45 flash chips from six companies with a variety of bit densities and manufacturing processes. Fitting the results to an exponential curve suggests a twofold increase in latency for every order of magnitude increase in density. In concluding, Grupp noted that we can improve density and cost, but performance and reliability will decline.

Nauman Rafique from Google asked why the authors consider the provided scenario to be bleak. Grupp replied that

consumers are accustomed to technologies improving, but we will not see this with flash. Michael Jadon from Radian Systems was optimistic that future precision improvements in voltage measuring would lower SSD latencies, but Grupp reiterated that the model only tracks current trends and does not include assumptions about future discoveries. David Rosenthal from Stanford University added that there is insufficient manufacturing capacity for flash to completely replace magnetic disks anyway, and that many of the limitations discussed apply to other technologies, such as Memristor. Abhijit Paithankar from VMware asked if the authors studied power consumption, but they had not considered this extensively. Kirk McKusick asked how the memory lifetime changes as we move to MLC and TLC. The author referred him to the paper, saying "It's a dramatic decline." TLC will only survive 500 program-erase cycles per block.

When Poll Is Better than Interrupt

Jisoo Yang, Dave B. Minturn, and Frank Hady, Intel Corporation

Jisoo Yang explained how the next generation of NVRAM will see interrupt overhead as a major source of latency. His work seeks to quantify the costs and to reduce them.

Conventionally, disks use a hardware interrupt to notify the scheduler when an I/O operation completes. The scheduler correspondingly wakes the thread waiting on that operation. This implies that between requesting and completing the operation, the requesting thread loses its context, freeing other threads in the application to do work. With a very low latency device, Yang argues, the overheads of handling this asynchronous I/O will start to dominate. It may be more efficient for the CPU to directly poll the device for completion. To test the claim, the team experimented with prototype hardware. They measured a DRAM-emulated PCIe SSD using the new NVM Express interface. They found that traditional interrupt-driven I/O had a 6.3 microsecond latency, while polling had only a 4.4 microsecond latency. Further, the asynchronous nature of interrupt-driven I/O left only 2.7 microseconds for an application to make any progress, limiting the benefit of asynchronous I/O.

John Groves of Dell Storage asked whether it would be better to have a polling loop in place of the entire interrupt handler. Yang clarified that each CPU in their implementation has a dedicated polling loop. Yan Li from UC Santa Cruz asked how the results should be interpreted in light of the preceding talk, "The Bleak Future of NAND Flash Memory." Yang responded that he expects the next-generation processes to move away from NAND and to make significant improvements in performance. Kai Shen from the University of Rochester noted that even with polling, OS overhead is substantial. He wondered if it is worthwhile to remove the OS completely, as is done today with GPUs. Yang said it was an interesting and likely effective approach, but that it would break the block interface, which has value. Steven Swanson from UC San Diego asked if the authors see potential benefit in increasing concurrency by using a polling thread with multiple outstanding requests. Yang believed the results would depend on the application logic. In some cases it might benefit, so it's a scenario worthy of further consideration.

Back It Up

Summarized by Yiqi Xu (yxu006@cs.fiu.edu)

Characteristics of Backup Workloads in Production Systems

Grant Wallace, Fred Douglis, Hangwei Qian, Philip Shilane, Stephen Smaldone, Mark Chamness, and Windsor Hsu, EMC Corporation

Fred Douglis started by highlighting some special characteristics of a backup system, e.g., it stores the data using aggregation instead of small files. Other characteristics include that backup data is replicated and that the backup data on a weekend usually has full 100 GB tar-type, while workdays have incremental 1 GB tar-type. He pointed out that analysis of primary storage abounds, while there is little characterization of backup systems. Their work can validate past design decisions using more extensive data and provide data for future analyses. The work is also motivated by the fact that it is predicted that there will be eight exabytes of data on disk-based, purpose-built backup appliances by 2015. Their two-pronged analysis covers a study of both breadth and depth, with statistics from over 10,000 systems and using detailed metadata traces from several production systems storing almost 700 TB of backup data.

Fred compared backup file size to primary storage file size. The former is orders of magnitude larger, so that traditional optimizations do not work for backups. Backup files also have many fewer files and directories, as well as flatter hierarchy because of many files per directory, and backup systems also use catalogs. The weekly churn is around 20%, so the system should be able to reclaim data on a regular basis. That's why deduplication helps. Primary data deduplication is reported to be 3x–6x, while backup data is >60x dedupe for some, 384x max. He went on to sensitivity analysis of chunk size and cache size. With many different kinds of data sets, he proposed merging chunks to analyze deduplication rates across a range of chunk sizes without having to access the whole content. They used content-defined merging and considered the overhead of metadata with smaller chunks. The rule of thumb is 15% better deduplication rate for each smaller power of 2 in chunk size, but about 2x the metadata. The best deduplication chunk size is 4 KB, and 8 KB consid-

ering maintenance and cleaning. For caching, they proposed replaying traces with varying cache sizes and reported results on the warm cache. The results show that for writes, chunk-level LRU caching needs large chunks to be effective for writes and that container-level LRU caching works well. Read cache behavior is similar but for much larger caches, due to data caching.

Fred covered the related work on deduplication and data characterization and concluded that high churn means throughput must scale with primary storage capacity growth. Backup systems are very different from primary systems. They need high locality and deduplication for hit rate high performance; 8 KB is a sweet-spot chunk size.

Dutch Meyer asked if 8 KB chunk average is an effective average they would measure or a statistical average they expect to get (it makes a difference if chunks break on zeros very often). Fred said that they are actually pretty close, that their system doesn't do anything special about zeros, and that if all zeros on a block define a block boundary, then it's going to cause a problem. Primary storage often has many files whose last chunk is smaller than the rest. If we get TBs of data on one file, then the last chunk is just noise. Dutch Meyer did set chunk boundary on zeros when he did his work and found it interesting that EMC doesn't. Dutch said it's really dependent on what the average means, and they took the discussion offline. Geoff Kuenning asked if data is quite anonymized and can they have it. Fred replied that they cannot make any promises and cannot join in a repository, at least for the foreseeable future, but they welcome interns to join them and have access to the data and work on it. Someone asked whether storage speed depends on chunk size. The answer was no. Arkady Kanevsky from Dell asked, if database churn is very quick in the dataset, is that characteristic or anomalous? Fred reviewed the slides and said it's an indication of how long the user would keep the backup, and it's really a choice of the user. John Groves asked if they are chunking with bins and Fred answered yes.

WAN Optimized Replication of Backup Datasets Using Stream-Informed Delta Compression

Philip Shilane, Mark Huang, Grant Wallace, and Windsor Hsu, EMC Corporation

Philip Shilane started his presentation by showing the demands and challenges in improving offshore replication performance. Afterwards he talked about his idea and demonstrated an example of deduplication, with delta compression sketches matching similar chunks. He compared his work by searching through several sketch index options (full index, partial index, and stream-informed cache) and analyzing their advantages and disadvantages. He further exemplified his idea by animating a discovery of similarity in data.

Philip compared the two approaches, replication with deduplication and replication with deduplication and delta compression. He discussed the properties of stream-informed delta compression, its pros being fast compression of data and small memory footprint, and cons being dependence on locality and cache, and some resource cost. The data set he used is very different from the previous one: multiple months' backups with varying sizes. The results from delta compressions are shown on multiple datasets, compared with a full index simulator. The results show that two super features are better than an index with one. More features do not necessarily improve compression, because of a fixed cache size. The results also show that one feature compression is within 14% of using an index. As a result, delta improvement is from 1.8x to 3.1x and the effective network throughput is 1–2 orders of magnitude faster than the old approach without delta compression.

Overhead and limitations were also discussed, with more space requirement per chunk and more CPU and I/O consumption on the source and destination. The sketching is also claimed to slow down the writes for non-duplicates by 20% and scales linearly with the number of streams at the destination. They also discussed compression loss because of shared caching size and the real case results from customers. Philip concluded his talk by listing the related works and stating that delta locality closely matches deduplication locality for backup datasets; in addition, the work has low cost and good scalability, and it allows customers to protect twice as much data by moving it across a WAN. Cristian Ungureanu from NEC asked why 3.1x compression improvement results in 1–2 orders of magnitude in network throughput. Philip answered that it's due to the performance variation in non-delta compressions.

Power Consumption in Enterprise-Scale Backup Storage Systems

Zhichao Li, Stony Brook University; Kevin M. Greenan and Andrew W. Leung, EMC Corporation; Erez Zadok, Stony Brook University

Zhichao Li claimed that disk backup is a prime candidate for power management, but there is no previous power measurement research and so assumptions are often made that disks will dominate power. The authors measured four enterprise backup controllers and two kinds of enclosures. They used a power meter for accurate measurement while excluding other factors such as networking, cooling, and internal subcomponents. First, Zhichao presented numbers for idle power consumption—when deduplication is being performed, CPU and RAM cost power. He used watts per TB to measure the different models and found that newer controller models are more efficient. The case for enclosures is similar. He concluded that deduplication saves space, because it saves

hardware such as the controller/enclosure and networking devices. It also reduces disk I/Os. Zhichao went on to disk power management—spin down/power down of disks has limited savings on power consumption. Other components in the enclosure drain more power. Disk spin-down at scale also demonstrates that in order to save power, many enclosures in a controller need to spin down their disks. He then looked at power proportionality in the controller. The results show that power varies more by model than by workload, because the controller consumes more power than needed. And in the enclosure the percentage power increase is less than the workload change, which proves again that controller/enclosures consume more power than disks.

Zhichao's conclusions were: (1) controller/enclosures are power hungry, (2) current systems are not power proportional, and (3) new hardware is more efficient. Future work will focus on aged backup, primary storage, CPU, and RAM power consumptions when built-in sensors are available.

Brent Welch from Panasas asked how realistic it is for the controller to populate 50 enclosures, because the controller also has to do RAID to protect drives. At what point does the deduplication controller spend more time on RAID rebuilds than on decompression? Will they really recommend that customers load 50 enclosures? Zhichao replied, it depends on whether the customer wants larger capacity with lower power cost, and how much customers are willing to pay.

File System Design and Correctness

Summarized by Yiqi Xu (yxu006@cs.fiu.edu)

Recon: Verifying File System Consistency at Runtime

Daniel Fryer, Kuei Sun, Rahat Mahmood, TingHao Cheng, Shaun Benjamin, Ashvin Goel, and Angela Demke Brown, University of Toronto

Awarded Best Paper!

Daniel Fryer presented work on an online filesystem-checking layer between the file system and block layers. He showed us that metadata is important while fragile in file system consistency. And metadata is more error-prone because of file system bugs and will cause data loss. Current solutions rely on the correctness of file systems such as journals, checksums, and RAID. Offline checking is often used, but it is slow, requires taking the file system offline, and produces error-prone repair.

Daniel's team proposes making sure that every update results in consistency. However, consistency properties are global and may require a full scan run; furthermore, fsck at every write is not possible. Thus, fast, local consistency invariants are introduced to keep data consistent before it becomes persistent. Daniel demonstrated an example in ext3, where the block bitmap and the block pointer should agree with each other. He further explained that the check happens on the transaction boundary just before the commit block reaches the disk. The design depends on the interpretation of metadata and comparison with the old version, without depending on the agnostic file system. He proposed an interface to invoke for different types of file systems. The write cache is used for delaying the update, and the read cache is for storing the hot cache of the read metadata.

Daniel discussed the evaluation of the implementation, detection effectiveness, and performance overhead. They simulated metadata error corruption by injecting wrong metadata before it was written. The errors they catch are inclusive of all the errors found by e2fsck except for a special flag that isn't being used in ext3. Cache size also affects one of the benchmarks evaluating throughputs, because the 128 MB cache size takes away some cache from the system.

Wenguang Wang from Apple asked what happens after catching an inconsistency; is stopping all I/O an option? Yes, they hold and fail stop. Several other options are possible, including stopping all writes, remounting read-only, taking a snapshot and continuing, and micro-booting the file system or kernel. Ethan Miller from UC Santa Cruz asked what kind of issue to expect when delayed commit is implemented in the FS. Daniel responded that ext3 also group commits with 5–10 thousands of blocks at a time. Keith Smith from NetApp asked for advice for future file system writers to make checking easy. Daniel said they like Btrfs, with back pointers that require less data to track; he also recommended figuring out and writing consistency problems in a declarative language like SQL. Someone asked if this kind of delay is tolerable in synchronous, production workloads with increased latency for commit block; they took the discussion offline. Atul Adya from Google asked if they considered applying this technique to other applications such as distributed file systems. Daniel said that they thought about this and will probably find transaction models and distributed invariants. Chris from Nebula asked why they do this work in the block layer. He maintains a subset of this kind of code for xfs within the file system. Daniel admitted that it is more practical for real systems this way, but placing the function on the block layer also has the benefit of taking it out of the kernel and placing it with the hypervisor if they don't trust the operating system, and that filesystem format doesn't change quickly.

Understanding Performance Implications of Nested File Systems in a Virtualized Environment

Duy Le, The College of William and Mary; Hai Huang, IBM T.J. Watson Research Center; Haining Wang, The College of William and Mary

Duy Le started by pointing out an inadequacy in the investigation of impacts of nested file systems. Existing literature

exists around I/O scheduler, storage allocation, and virtualized FS on the virtual machines. However, assumptions made on one layer of file system may hurt two-layer schemes. The authors combine six file systems (ext2, ext3, ext4, reiserfs, xfs, and jfs) as host and guest file systems to find the best combination with varied I/O behavior and interactions.

In their setup, they partitioned the physical disk into equal partitions and formatted six of them to six different file systems, which in turn used a flat file to act as disk image for a virtual block device. The last partition was used as a direct block device for baseline measurement.

The results show that guest file system and host file system choice are bi-directionally affecting each other in performance. While writes are more critically affected by the additional layer, read sometimes can achieve even better performance. Latency is sensitive to nested file systems. Duy then zoomed in on some specific combinations of guest and host file systems for detailed analysis on reads and writes. He showed some findings, including the effectiveness of guest file system block allocation, I/O scheduler's effectiveness on the guest file system, and journaling performance impact. Finally, he listed five pieces of advice for file system choice/tuning for different workloads and circumstances.

Erez Zadok from Stony Brook University asked why they did not shuffle the host file systems on each of the partitions. This could result in zone-constant angular velocity, which is said to have up to 25% marginal variance on performance. Duy replied that they have demonstrated a less than 5% performance difference and determined that it is a negligible factor in the evaluation, since 42 combinations will cost a lot more in time. Zadok then asked whether it will be different when they put different kinds of workloads/access patterns on the partitions. John Groves from Dell asked whether the container file is pre-allocated (yes) and did they use direct I/O to avoid upper-level I/Os going to page caches of the underlying file system (yes). Ric Wheeler from Red Hat commented that NOOP is often used in the upper level in a virtualized environment. Duy said that guest CFQ / host deadline was the best combination they found, so they tended to use this setting. Dutch Meyer (University of British Columbia) asked whether images are raw (yes) and can the findings/approaches be generalized at the disk management layer. The topic was taken offline. Was disk flushing disabled for accurate measurement? They made sure all caches got flushed.

Consistency Without Ordering

Vijay Chidambaram, Tushar Sharma, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau, University of Wisconsin, Madison

Vijay Chidambaram pointed out that crash inconsistencies are caused by ambiguity about logical object identity. He showed a crash scenario in which two files claim a shared block. He then introduced No-Order File System (NoFS), which uses back pointer-based consistency, where owners of objects are found through the back pointer; the important assumption is that object and back pointer are written atomically. He revisited the crash scenario to show how using a back pointer works. He then showed how allocation structure consistency is maintained. The creation of new objects can proceed without complete allocation information. The validity bitmap is used to track checked objects. He elaborated the case with a scenario determining allocation information. Implementation details include two threads responsible for metadata and data scan in the background. Note that scheduling of scans can be configured while idle or periodically.

In evaluating NoFS, Vijay answered three questions: is it robust? how much is the overhead? and does the background scan lower performance? They put a pseudo device driver between the file system and disk to discard writes to selected sectors to simulate crash. As a matter of fact, NoFS detected all the inconsistencies, and orphan structures were reclaimed. There proved to be minimal overhead in terms of throughput, and ext3 demonstrates lower throughput in cases where there are order points in writes. However, scan reads interleaved with file system I/Os really affect throughput, and accesses on non-verified objects have penalties, such as stat. Vijay concluded that trust is implicit, and removing it is key to robust, reliable storage systems.

Peter Macko from Harvard asked about the memory overhead of this approach. Vijay replied that one extra block for the bitmap is acceptable. Bill Bolosky from Microsoft Research asked how big the file system is. Vijay answered that it's a 50 G partition with 1–5 GB of data. Bill responded that a 3 TB surface scan takes 4–5 hours. How does this system work when it's close to full? Does it wait a long time for writing (finding a free block)? He mentioned the work he was doing on distributed file systems, which are often full. Ashvin Goel from the University of Toronto asked if this approach works for all disks. Vijay answered, as long as the back pointer is atomically written together with data block on the disk, it can be achieved on the device. Someone asked where the back pointer is stored. Vijay answered that in the implementation it is inside the data block. But when out of band area can be used, they can store it there. Keith Smith from NetApp commented that if a file system grows big enough, it will not be able to handle the extra block. He asked if files created on disk can be in the same order as created in the applications. Vijay said no, since it's not the consistency they provide. Dave Anderson from Seagate Technology recommended that Vijay apply this technique to other problems. Vijay said yes, it's applicable to hierarchy problem domains. Brad Morrey from HP Labs asked about the extra CPU cycles

this approach introduces. Vijay answered that there is no noticeable increase, because the check occurs between the disk and memory. Someone from Seoul National University asked how the back pointers are removed when deleting files. Vijay said their approach is lazy deletion. He talked about having version number consistency if necessary. Vijay said that only bitmaps are kept in memory and they don't keep any structures, so there is no need to free many objects.

Flash and SSDs, Part I

Summarized by Dutch Meyer (dmeyer@cs.ubc.ca)

Reducing SSD Read Latency via NAND Flash Program and Erase Suspension

Guanying Wu and Xubin He, Virginia Commonwealth University

Guanying Wu presented his research on making the program-erase cycle of NAND flash memory suspendable. To rewrite a page of NAND flash memory, the drive must complete a program-erase cycle, which first erases the page and then reprograms it. The erase operation is performed with a long pulse of erase voltage to expel electrons from the cells. The subsequent programming requires a series of charges in which a short pulse is attempted, then tested. This programming is retried with successively higher voltage pulses until it is successful. This process can be 10x–100x slower than a read, which only requires measuring the cell voltage. The problem is that a program-erase cycle blocks out all reads to the chip where the block exists, causing higher latency.

To improve read performance, Wu and his team developed a method for making the program-erase cycle suspendable. This allows read requests that arrive during a lengthy program-erase operation to be quickly fulfilled and the rewrite to be resumed later. This suspension may occur at different points in the program-erase cycle, each requiring a different mechanism. If suspension is needed during a program operation, suspension occurs between program pulse and verify operations. During the erase cycle, the erase operation is stopped and the duration remaining is noted. The erase can continue when the interrupting read request completes. In evaluating the system, Wu found that write latency increased by a few percent, which he considered trivial. Read latency decreased by 50% or more. He concluded that suspending the program-erase cycle for reads is a feasible solution, and one that significantly improves read performance.

Umesh Maheshwari from Nimble Storage noted that this approach seems to work best for lightly loaded systems, and Wu agreed. Dave Anderson (Seagate Technology) asked if flash lifetimes are degraded because of the extra wear involved in restarting program-erase cycles, but Wu had not performed that experiment at the time of the presentation.

Optimizing NAND Flash-Based SSDs via Retention Relaxation

Ren-Shuo Liu and Chia-Lin Yang, National Taiwan University; Wei Wu, Intel Corporation

Ren-Shuo Liu presented a technique for optimizing flash performance by opportunistically relaxing the requirements for longterm data retention. He noted that there are two main reliability specifications in flash: bit error rate and data retention time. The latter specifies how long the data should stay durable on stable storage. Error correcting codes (ECCs) are used to ensure that both criteria are met, but as flash density increases, the raw reliability degrades. This forces manufacturers to slow down writes to mitigate the worsening bit error rate and to strengthen ECCs.

However, data retention time of one year is overly conservative for the majority of data. Liu's team proposes a retention-aware FTL that initially relaxes the data retention specification to two weeks, and if data is not overwritten in one week, it can be reprogrammed with the stronger retention policy. The result is that long-lived data is eventually given the longer retention policy, but data overwritten within a week can use a weaker ECC to speed up writes by a factor of two.

To evaluate the approach, Liu used 11 workloads gathered from Microsoft Research Cambridge (MSR-C) and synthetic workloads, including TPC-C and Hadoop benchmarks, to estimate the average lifetime of a block of data. As a point of reference, in the MSR-C workload 86% of writes are overwritten in less than one hour. These workloads were evaluated on disksim 4.0 and SSDsim, using the retention-aware FTL design. The results suggest that a 2 to 5.7-fold improvement in performance is possible.

Several in attendance, including Sam Noh from Hongik University and Dave Anderson, asked Liu to clarify the durability assurances of long-lived data. Liu explained that background processes always convert long-lived data to normal retention mode, so there is no longterm relaxation of durability requirements. Another participant asked if any effort has gone into understanding the effects of the garbage collector. Liu responded that only writes originating from the guest use the weaker ECC. Data movement due to garbage collection always uses full ECC. Liu was also asked if the same performance improvement could be had if the controller was made more powerful, to accommodate stronger ECC, but that conversation was taken offline. Geoff Kuenning from Harvey Mudd College asked Liu to clarify the methodology around measuring the cleaning that occurs due to long-lived data being moved to a high retention cell. Liu assured Kuenning that the cleaning process was included in the simulation.

SFS: Random Write Considered Harmful in Solid State Drives

Changwoo Min, Sungkyunkwan University and Samsung Electronics; Kangnyeon Kim, Sungkyunkwan University; Hyunjin Cho, Samsung Electronics; Sang-Won Lee and Young Ik Eom, Sungkyunkwan University

Changwoo Min presented a new file system for solid state disks (SSDs) that's designed to address two fundamental limitations of these devices—random write performance and limited lifespan. As SSD technology matures, lifespan is a concern because each bit added to a memory cell decreases the number of accurate rewrites by an order of magnitude. Meanwhile, random write workloads significantly degrade performance and further shorten the lifespan of SSD.

Min described SFS, which is a file system specifically designed to remedy these problems. The file system employs a log-based design, which is suited to the unique characteristics of SSDs. By writing in a log structure and carefully grouping requests to match the size of an erase block, SFS effectively transforms random writes on the SSD into better-performing sequential writes and removes most internal fragmentation. When writing, it groups data according to hotness, so that future updates are less likely to require moving otherwise cold data. As a metric for hotness, SFS tracks write count and divides by the age of the block. It groups blocks together using an iterative refinement technique inspired by k-means clustering.

Min's team evaluated their system on three classes of SSD, using two synthetic workloads—TPC-C and a workload collected by UC Berkeley—and compared the results to ext3 and Btrfs. The results showed that SFS requires up to 6.1x fewer erase operations. The system is effective at making segment fullness bimodal, with more segments being either completely full or completely empty. As future work, Min is applying the file system to magnetic disk-based storage. His early investigations show some "promising results."

Min was asked if indirection at the file system level could cause additional write amplification. He answered that this is not what they have seen. In practice, total block erase counts are lower. Seungjae Baek from the University of Pittsburgh noted that grouping according to hotness has a long history and asked about comparisons to other schemes. Min directed Baek to comparisons in the paper. Umesh Maheshwari from Nimble Storage asked how the designers matched their segment size to that of the flash drive, and was particularly concerned about misaligned blocks. Min acknowledged that misalignment degrades performance, but the researchers had selected the file system parameters used by directly measuring the erase block size of their flash drives. Did Min's team have any plans to productize the file system? They were still discussing that possibility. Finally, one attendee asked whether the design depends on deferring writes until a full erase block is available, and cited sync operations in TPC-C as an example. Min responded that in sync operations, the system works as best effort.

Poster Session I

Summarized by Dulcardo Arteaga (dulcardo@gmail.com)
Only posters that were not represented by papers are summarized here. See static.usenix.org/events/fast12/poster.html for PDFs and descriptions of all posters.

CSPE: Cloud Storage Provisioning Decided by Rate of Return and Workload Characteristics

Jianzong Wang, Rui Hua, Changsheng Xie, Jiguang Wan, Yanjun Chen, Peng Wang and Weijiao Gong, Wuhan National Laboratory for Optoelectronics

This project presents a model that evaluates current workload on a cloud and its tendency to determine the benefit of purchasing/leasing new disks. They used the Internal Rate of Return (IRR) model used in economics to solve the "purchase or not" problem. They also used one module to detect workload peaks and another to trace the workload.

Reliable Energy-Aware SSD-based RAID-6 System

Mehdi Pirahandeh and Deok-Hwan Kim, Inha University

This project presents an approach for periodic estimation of reliability and energy consumption and a model of RAID6 that saves energy. The idea is converting pages into packages to reduce the amount of work during writes. Their evaluation shows that there is improvement in the energy consumption.

InnoDB DoubleWrite Buffer as Read Cache Using SSDs

Woon-Hak Kang, Gi-Tae Yun, Dong-In Shin, Yang-Hun Park, and Sang-Won Lee, Sungkyunkwan University; Bongki Moon, University of Arizona

Woon-Hak Kang presented this work to extend and move the double write buffer of InnoDB to an SSD to exploit the capacity and low latency of this kind of device. Besides the functionality of writing dirty pages to guarantee atomicity, they propose using the double-write buffer as a cache for random reads, consequently improving the performance of reading and for writes. The evaluation shows a significant improvement in performance for reads but not for writes when comparing the use of HDD to SSD.

Mitigating the Network Impact in Large Scale DFSs

Gustavo Bervian Brand and Adrien Lébre, Ecole des Mines de Nantes

This project evaluates the performance of different distributed file systems based on different topologies, comparing when data servers/metadata servers are located behind a

WAN/LAN. This evaluation attempts to demonstrate that there is a need to include the factor of topology when designing a distributed file system.

Their experiments compared a variety of topologies with different numbers of nodes, and their performance, based on different configurations, varies considerably, due to the overhead in the network traffic.

vPFS: Bandwidth Virtualization of Parallel Storage Systems

Yiqi Xu, Dulcardo Arteaga, and Ming Zhao, Florida International University; Yonggang Liu and Renato Figueiredo, University of Florida; Seetharami Seelam, IBM T.J. Watson Research Center

Yiqi Xu proposed vPFS, which adds to existing parallel file systems the ability to differentiate I/O requests from different applications, then meet per-application quality of service. This approach was implemented on top of PVFS, which is a user-level distributed file system. A proxy was created to intercept the I/O traffic and tag it according to the applications, and a scheduling algorithm is applied at that point to meet the quality of service.

Experiments show that using different applications with different QoS can meet application requirements without generating overhead in the I/O.

OS Techniques

Summarized by Daniel Fryer (dfryer@cs.toronto.edu)

FIOS: A Fair, Efficient Flash I/O Scheduler

Stan Park and Kai Shen, University of Rochester

Stan described the increasing adoption of flash-based storage, and how there's been very little work in I/O scheduling for flash. In particular, synchronous writes have been a bottleneck for I/O, and they continue to be on flash devices. Then he described the characteristics of flash devices that distinguish them from disks, particularly the lack of seek latency, the large erase granularity, the need to erase before write, and variations between vendors. He then gave an example of why "fairness" in flash I/O scheduling matters: during conflicting reads and writes both response time and variance increase greatly. Each vendor's devices react differently. This was also demonstrated for requests issued in parallel—some devices give better results than others.

This led to the development of FIOS, with fairness as a first-class concern. They try to use fairness but also efficiency by exploiting I/O parallelism, and also use I/O anticipation (delaying I/O briefly) to help achieve this. The approach of FIOS is timeslice management: give tasks equal amounts of time to access the device (possibly non-contiguously).

Timeslices are grouped into epochs; an epoch ends when all tasks have spent their timeslice or there are no pending I/O requests. Because reads are faster than writes, reads are penalized more by interfering writes. The authors introduce interference management by servicing reads quickly and delaying writes until the reads complete. This minimizes the opportunity for interference.

To exploit I/O parallelism while still respecting their timeslice management, the authors require some cost accounting method for parallel requests. They have two models: a linear cost model, and probabilistic fair sharing. The linear cost model calculates I/O cost as a function of request size, calibrated by times to service reads and writes of different sizes. The probabilistic fair sharing model tries to estimate the amount of concurrency occurring. They used the linear cost model for the results in the rest of the talk.

Anticipatory I/O was used on disks to improve performance hits due to deceptive idleness. It's not necessary for flash performance, but they use it for fairness. Deceptive idleness can cause an epoch to end early, robbing some process of the remainder of its timeslice. Also, a write issued immediately before a group of read requests is bad. The question is how long to wait, without wasting valuable I/O time. Their default is to wait for half of an average request service time.

They evaluated fairness by measuring the I/O slowdown ratio—the amount a request's response time is degraded compared to running alone. For N tasks, proportional slowdown means that each task should experience no more than a factor of N slowdown. They show that the NOOP, CFQ, and SFQ(D) schedulers slow down reads dramatically, while writes are faster than proportional; FIOS achieves fairness, with both reads and writes faster than proportional slowdown. Quantum-based scheduling is fair, but relatively slower. Stan showed how FIOS also performs well under asymmetric read/write loads. Running SPECweb and TPC-C simultaneously showed that FIOS maintained fairness under real workloads. Finally, performance on a low-power CompactFlash system showed better read efficiency than the other schedulers and fair write performance.

Stan reiterated that fairness was their primary concern. I/O anticipation is important for fairness, even though it's not important for pure throughput. The I/O scheduler must be robust in the face of differing flash architectures. The authors believe that the FIOS approach to fairness might also be applicable to other domains such as virtual machines and the cloud.

Geoff Kuenning (Harvey Mudd) asked what their definition of fairness was on the slide showing asymmetric I/O. Their measure was equal latency—how much each task is slowed down. Geoff thought that this was a matter of opinion, but

didn't want to push the matter. Someone from Google asked whether they planned to look at ticket-based schedulers, where tickets are issued proportional to I/O sizes, to give them parallelism without anticipation. Stan explained that these policies aren't specific to FIOS—FIOS was the artifact that came out of looking at these policies—and that they could try a ticket-based approach. Vasily Tarasov (Stony Brook) wanted to know whether they ran experiments with different priorities assigned to tasks. The authors had not, but Stan suggested that they could do scheduling within each priority class, or hand out different-sized timeslices.

Shredder: GPU-Accelerated Incremental Storage and Computation

Pramod Bhatotia and Rodrigo Rodrigues, Max Planck Institute for Software Systems (MPI-SWS); Akshat Verma, IBM Research—India

Pramod Bhatotia started with a fundamental problem: given that the total volume of data is growing rapidly, how can we efficiently store and process it all? One major technique is to eliminate redundancy. Redundancy elimination is expensive, however, and is a three-step process. First, a file is broken into chunks, then the chunks are hashed, and finally the hashes are matched to establish whether or not a duplicate exists. "Content-based chunking," introduced in SOSP '01, uses a sliding window over a file rather than fixed chunks. This can keep boundaries stable under small insertions or deletions in the data. Unfortunately, it is very CPU intensive, which can be a bottleneck.

Content-based chunking throughput on a multicore machine is about 0.5 GBps, but about 1 GBps with a standard GPU-based design. This is a 2x improvement, but still not good enough—the target I/O bandwidth they're trying to support is 2.5 GBps! The reason for the performance gap is because GPUs are designed for compute-intensive workloads, not data-intensive workloads.

Buses run from host memory to CPU, from the CPU to GPU (PCI), and inside the GPU, which is divided into a set of multiprocessors with local memory and a global pool of shared memory. The CPU can only access this global memory, and not the private memory of the compute units. First, data is transferred to device memory, then the CPU launches threads on the GPU, which loads data from global GPU memory into its local memory for fast access, and eventually pushes results back to the host.

There are several scalability problems here. The cost of data transfer across the PCI bus is comparable to the time spent in the chunking kernel. Shredder pipelines data transfer by dividing GPU global memory into two portions, and loads one while chunking is being executed on the other. The second problem is memory access conflicts on the device itself. For speed, data needs to be fetched from global GPU memory into the local processing unit memory. Global memory is divided into interleaved banks, and threads accessing the same bank simultaneously cause a conflict, leading to serialization. Shredder's solution is to coordinate threads so that they cooperate to fetch data for a task from separate banks, leading to parallel bank access. Then the threads can work in isolation on device local memory.

Shredder was implemented using C++ and CUDA, and benchmarked on an NVIDIA Tesla c2050 hosted on a 12-way Xeon. The basic GPU approach achieved 1 GBps. Using pipelined CPU->GPU transfers, this can be improved to 1.75 GBps. Finally, using the coalesced threads loading local processing unit memory, they achieve 2.25 GBps. Pramod then presented a case study on incremental MapReduce, where some input has changed; they want to recompute along the path from this changed input using the other unchanged intermediate results. The problem is that using fixed-size chunking would throw all the chunk boundaries off if data is inserted or deleted, so they use content-based chunking to partition data before running the MapReduce process.

Someone from EMC BRS asked what their baseline multicore performance was, since OpenSSL gets 350 MBps on a single core for SHA1. Did they need all 12 cores to get 500 MBps? It seems slow for a multicore, and it was suggested that they could get the same performance by tuning their CPU implementation. Pramod disagreed, but discussion was deferred. Brent Callaghan (Apple) wanted to know whether they did hashing in the GPU as well as the chunking. Pramod clarified that they are only finding chunk boundaries, although in theory they could do the hashing as well. Someone from UC Santa Cruz wondered what the difference was between this and scientific computation, since there has been work done on using GPUs for scientific workloads. Pramod differentiated the two by suggesting that scientific applications are often N^3, N^4 while chunking is linear, so it's all about transfer bandwidth.

Adding Advanced Storage Controller Functionality via Low-Overhead Virtualization

Muli Ben-Yehuda, Michael Factor, Eran Rom, and Avishay Traeger, IBM Research—Haifa; Eran Borovik and Ben-Ami Yassour

Avishay noted the need for new functionality in storage controllers (e.g., deduplication or compression) that has already been implemented elsewhere. The conventional approach has been to port the functionality from its original environment onto the storage controller itself. This has the advantage of low overhead but incurs a high engineering and maintenance cost. Another approach is to perform the function on a separate machine, which he calls the "gateway" approach. This avoids the cost of porting the software, but incurs a runtime

performance cost as well as the additional cost of the gateway hardware.

A third way is a hybrid approach: running a VM on a storage controller. Unfortunately, virtual machines have a bad reputation for overhead. Storage controllers are special-purpose devices with finely tuned resource control. Virtual machines provide a large number of features, not all of which are needed on a storage controller—they need the fault isolation and the separate environment, but they don't need resource-sharing, the ability to overcommit, or migration. So they thought that perhaps they could customize virtual machines to make them suitable for use on storage controllers.

Avishay defined external communication as the I/O between client and VM; internal communication is the communication between the VM and the controller. In their approach, the I/O interfaces are directly assigned to the VM, although servicing interrupts and I/O completions require a hypervisor context switch. To communicate between the VM and the controller they use a virtual I/O block device built on top of shared memory, but it also requires hypervisor switches to handle the interrupts and I/O completions.

Their main approach to avoiding the latency of the hypervisor is polling. They run a polling thread on the guest VM which polls the NIC, avoiding the need for an interrupt. The block request is put into shared memory by the VM; the host detects this request by its own polling thread. When the request on the controller is complete, the host process puts the completion in shared memory, where it is detected by the original polling thread on the VM, finishing the I/O path with no interrupts. They also statically allocate CPU cores and memory, since they can establish resource usage parameters beforehand. The VM is configured to poll when idle instead of sleeping, since they don't need to share with other VMs. They minimize memory management overhead by backing the VM's memory with HugePages.

To benchmark, they configured two servers, each with a pair of quad-core 2.9 GHz Xeons and 16 GB RAM. One server functioned as a load generator, the other served as an emulated storage controller. They compared their VM-based solution to a "bare metal" implementation with four cores assigned to the host. Storage was emulated by an 8 GB RAM disk to avoid the I/O bottleneck of a physical disk.

Their first evaluation is response latency during a ping flood. They show that with no polling, the bare metal solution takes 24 ×s, but the VM uses 89 ×s. On the other hand, with polling enabled both solutions take 21 ×s. On the Netperf benchmarks, guest and host polling show the best performance among the configurations they tried, except on TCP receive throughput, which they claim is because no real work is being done. They calculate that in the optimized case, they add only 6 ×s of latency to a random 4k synchronous write. They improve read throughput by a factor of 7 and write performance by 6 times. Finally, they present the incremental improvements of each of their optimizations, showing that after the work they've done they match the bare metal performance at the end. When they cut the controller down to four cores again, they take a performance hit because of the competing polling threads, but they can tweak thread priorities and affinities to overcome this.

Their conclusion is that virtual infrastructure can be used with near zero performance overhead. This provides the benefits of the high performance and low hardware cost of native integration combined with the shorter time to market and simpler development of the gateway approach.

Lakshmi Bairavasundaram (NetApp) wanted to know, if cores were assigned statically, how would they deal with situations where VMs were being supplied by multiple vendors? Avishay responded that you only need to configure it once, when you figure out what software you are deploying on your controller. Dutch Meyer of UBC asked if the presenter could comment on virtual storage appliances. Avishay said that his understanding was that they run extra functionality outside of the controller.

Mobile and Social

Summarized by Swapnil Patil (svp@cs.cmu.edu)

ZZFS: A Hybrid Device and Cloud File System for Spontaneous Users

Michelle L. Mazurek, Carnegie Mellon University; Eno Thereska, Dinan Gunawardena, Richard Harper, and James Scott, Microsoft Research, Cambridge, UK

In this talk, Michelle Mazurek presented a new file system for mobile/home networked devices through the use of new hardware components and a combination of storage system techniques. The goal of this file system, called ZZFS, is to provide spontaneous data access with good trust and control over data storage. Michelle first presented a user study from traces in the LiveMesh and Dropbox service; this study was driving the design of their ZZFS system. Key observations of this study include: (1) users are busy and want spontaneous response from the system, (2) users do not know their data needs a priori, and (3) users place/organize their data in a planned and reasoned manner.

Because battery life is a key concern on many mobile devices, ZZFS relies on an existing hardware component, the Somniloquy NIC, that provides on-demand network interface card wakeup with some on-board flash. One hardware assumption ZZFS made was that users rely on broadband connections at home with weak 3G-based connections on mobile devices.

ZZFS used a combination of well-studied storage systems techniques, including flat namespace metadata service, policy-driven I/O director service, and I/O offloading for effective power management. The authors built a prototype of their ideas and evaluated the performance of ZZFS's design for spontaneous and ad hoc data access.

Margo Seltzer (Harvard) asked about the few random slow requests in write latency. Michelle said the problem was probably with the WiFi router in the experimental setup. Jason Flinn (Michigan) said that centralized storage has advantages, but he agreed with Michelle' about it being hard to trust and asked for thoughts on how that could be improved. Michelle said that all trust-based issues are more about the experience; several companies have had bad experiences with user data and trust—this affects people's attitudes toward using centralized systems. John Berry (Riverbed) asked about the cache coherency policies in the system: for example, updates to the shared music repository. Michelle answered that if a device is on, the updates are sync'd and serialized. John asked whether there could be races in the middle of a song. Michelle replied that ZZFS relies on the Everest system at MSR (published few years ago) which uses serialization through the primary copy of the data.

Revisiting Storage for Smartphones

Hyojun Kim, Nitin Agrawal, and Cristian Ungureanu, NEC Laboratories America

Awarded Best Paper!

Nitin presented work on the performance of storage systems in smartphones. Mobile devices are becoming increasingly diverse in hardware, software, and application ecosystem. Most existing work has studied network and CPU performance; much work has also been adopted by system developers to make better use of these two resources. The authors studied the performanceof a suite of popular apps on a Google Nexus phone with Android OS. Although both the hardware and the systems software were commodity, the authors patched the OS with some measurement and monitoring extensions. To measure the storage I/O behavior, this work used different storage media devices (i.e., SD cards); this enhanced the study to be more device agnostic.

Several lessons emerged from this work. For SD cards, the results showed high disparity between random and sequential I/O performance; the device specifications are "bloated," because vendors report sequential speeds instead of slow random I/O speeds. Furthermore, the authors also observed that the storage device performance has not improved as much as the network speed performance over the past few years. In terms of applications, it was observed that although there are only half as many random writes as sequential writes, most apps write sufficient random data that the application performance is adversely affected. Application performance is also dependent on the quality of the storage media. In terms of systems software, applications target their writes either to a file system (in-memory, cached file system) or to a SQLite database. Both FS and DB behave differently; in particular, the synchronous writes used by many apps cause the DB writes to be much slower than FS writes.

Geoff Kuenning (Harvey Mudd) said that an Apple I/O study showed that most apps do a lot of synchronous I/O. Since you make similar observations, is it just that plain stupid apps are the problem? Nitin said that the common theme is the presence of App-OS modularity and interface boundaries—which is not good in all cases, particularly when performance is a victim of that modularity. Eno Thereseka (MSR) asked, does it really matter if storage is the bottleneck in end-to-end experience? Nitin replied that as the users of the phones and apps, they found that app performance is highly variable depending on how one uses the apps and phones; since all measurement is at the user level, they captured as much end-to-end performance as possible using black-box phones.

Serving Large-scale Batch Computed Data with Project Voldemort

Roshan Sumbaly, Jay Kreps, Lei Gao, Alex Feinberg, Chinmay Soman, and Sam Shah, LinkedIn Corp.

Roshan Sumbaly presented their large-scale batch processing system built on a key-value store. LinkedIn runs large Hadoop applications that need to bulk-load massive amounts of data in a system that is online and is processing active requests. LinkedIn relies on the Voldemort key-value store that was inspired by Amazon's Dynamo paper. In this work, LinkedIn developers extended Voldemort to overcome performance degradation due to index creation and mutation.

Extensions to Voldemort included incremental bulk loading, data error minimization and mitigation, and ease of use through configuration management. Two existing approaches, including a Hadoop-based insertion tool and multi-cluster deployment, failed due to performance interference and management complexity. Instead, the authors used Hadoop's parallelism and fault tolerance to build an intermediate table construction phase that relied on easy rollback using versioned datasets. They also added simple rebalancing protocols to drive changes in the data-to-memory ratio to mitigate bottlenecks due to memory utilization. Results from their production system show that LinkedIn's extensions help run large data pipelines while maintaining sub-5 ms latency for active users.

Konstantin Shvachko (eBay) asked how rollback worked with new versions. Roshan said that rollback is more than

just symlink repointing. It also involves closing an mmaped index, updating the current versions, and then updating symlinks. Rollbacks are worst-case scenarios to handle production problems. Dan Peek (Facebook) asked, why not have one large ring instead of many small rings? Roshan replied, because there is a chance that rebalancing work may increase for a single ring.

Work-in-Progress Reports (WiPs)

Summarized by Michelle L. Mazurek (mmazurek@cmu.edu)

Generating Realistic Datasets for Deduplication Analysis

Vasily Tarasov and Amar Mudrankit, Stony Brook University; Will Buik, Harvey Mudd College; Philip Shilane, EMC Corporation; Geoff Kuenning, Harvey Mudd College

Research and industry have developed many different deduplication protocols. Comparing them is difficult because evaluation depends so heavily on the dataset used. What is needed is a benchmarking dataset that is large, realistic, versatile, easy to distribute, and with parameters that are easy to tune. This work attempts to develop such a dataset by observing and emulating the way that real file systems mutate over time.

Disk-Failure Injection Framework for Fault-Tolerant Systems Research

Yathindra Naik, Mike Hibler, Eric Eide, and Robert Ricci, University of Utah

It is important to understand how modern complex storage stacks will behave in the face of disk failures. This work builds a framework for injecting disk errors for testing. The framework should be realistic, controllable, repeatable, scalable, and scriptable. Using Emulab, the presenters model delayed I/O, corrupt reads and writes, and sector errors. In progress: more realistic failure models that reflect the realistic frequency and distribution of these errors. A prototype will be available soon.

DS-RAID: Efficient Parity Update Scheme for SSDs

Jaeho Kim and Jongmin Lee, University of Seoul; Jongmoo Choi, Dankook University; Donghee Lee, University of Seoul; Sam H. Noh, Hongik University

Current SSDs provide low reliability, a high error rate, and a limited erase count, with multi-level cells exacerbating the problem. Current approaches to applying RAID5 (striping) to SSDs have limitations related to small writes and the inability to write new data until a stripe becomes full. Parity pages must be written too frequently, increasing wear. This work uses dynamic striping to solve these problems. Stripes are constructed based on arrival order, containing non-consecutive block numbers. Sub-stripe parity is used for write requests smaller than the stripe size. Evaluation shows DS-RAID results in fewer extra reads and writes, with less cleaning cost, than standard RAID5.

The Peril and Promise of Shingled Disk Arrays (How to Avoid Two Disks Being Worse Than One)

Quoc M. Le, JoAnne Holliday, and Ahmed Amer, Santa Clara University

Shingled disks promise to increase storage density for disk drives, but must be used carefully because updates to written tracks may overwrite neighboring tracks. This work evaluates the behavior of shingled disks when used in array configuration or when faced with heavily interleaved workloads from multiple sources. There are three evaluation workloads: pure (sequential workloads), striped (interleaved workloads), and dedicated (one workload per disk). As might be expected, more interleaving results in more disk activity as bands are relocated. Proper use of shingled disks may therefore require rethinking traditional disk array layouts.

A Unified Object Oriented Storage Architecture

Andy Hospodor, Ethan Miller, Rekha Pitchumani, Yangwook Kang, and Darrell Long, University of California, Santa Cruz; Ahmed Amer, Santa Clara University; Yulai Xie, Huazhong University of Science and Technology

This work presents a unified storage architecture based on object-oriented storage. The goal is to decouple metadata from data, allowing management of objects rather than blocks. This architecture can apply to a range of devices including magnetic, optical, SSD, tape, and even shingled disks. The presenters suggest that such an architecture should be designed from scratch, rather than attempting to extend SCSI. The architecture will provide typical methods such as read and write, as well as new methods including find, append, replicate, merge, and sort. An OO storage device should use a publish-subscribe model to allow operating systems to access these methods.

Challenges in Long-Term Logging and Tracing

Ian F. Adams and Ethan L. Miller, University of California, Santa Cruz

Long-term logs and traces are important, as some trends in system use aren't apparent in the short term. We have good tools for capturing log and trace data, but not for maintaining it over the long term: it's too much data, there are occasional hiccups in collection, and log formats change. The presenters propose periodically transforming older data to coarser resolution, so it takes less space and is easier to work with, while keeping fine-grained logs for particularly interesting events. They also suggest annotating logs and traces to indicate events like maintenance, nodes or processes going

down, etc. Another idea is to combine traces with snapshots of system state, so you can use the trace as a record of the change between snapshots. Finally, it's important to periodically check for format consistency and note anomalies and problems, so that log parsers don't break or, even worse, fail silently when processing a lot of log data.

Dynamic Block-level Cache Management for Cloud Computing Systems

Dulcardo Arteaga, Douglas Otstott, and Ming Zhao, Florida International University

Block-level network storage is used in cloud systems to provide VM storage and allow fast VM migration as well as VM availability. However, as cloud systems increase in size, there are scalability problems. This work uses dynamic, block-level client-side caching to improve performance at scale. This approach exploits data locality in VM data access, while supporting dynamic and flexible cache configuration. Each host contains one cache, which all the VMs on that host share. Within this cache are virtual caches for each VM so they don't interfere with each other. When the VM migrates, the virtual cache is flushed. Preliminary results show improved throughput using the IOzone benchmark.

CASE: Exploiting Content Redundancy for Improving Space Efficiency and Benchmarking Accuracy in Storage Emulation

Lei Tian and Hong Jiang, University of Nebraska—Lincoln

Storage benchmarking is very sensitive to content, with the same operations on different content potentially inducing very different performance results. As a result, benchmarks must retain data content. CASE aims to provide flexible, space-efficient, timing-accurate, and content-aware storage emulation for benchmarking. CASE is implemented using data deduplication over fixed-size chunks. Preliminary results indicate that CASE saves up to two orders of magnitude in storage space.

Trusted Storage

Anjo Vahldiek and Eslam Elnikety, MPI-SWS; Ansley Post, Google; Peter Druschel and Deepak Garg, MPI-SWS; Johannes Gehrke, Cornell; Rodrigo Rodrigues, MPI-SWS

Storage is complex, involving millions of lines of code, operating systems, file systems, drivers, etc. This complexity means vulnerability to bugs, viruses, and operator errors that threaten integrity, confidentiality, and durability. The presenters developed a trusted storage architecture that enforces user-provided policies for application objects like files. Policies may be based on user ID, hardware or software configuration, quota, time, or location, and govern the conditions under which objects can be read, updated, or deleted. Signed certificates attest to the object's properties, policies, and access history. Overhead for the implementation is expected to be below 3%.

Accelerating Data Deduplication by Exploiting Pipelining and Parallelism with Multicore or Manycore Processors

Wen Xia, Huazhong University of Science and Technology and University of Nebraska—Lincoln; Hong Jiang, University of Nebraska—Lincoln; Dan Feng, Huazhong University of Science and Technology; Lei Tian, University of Nebraska—Lincoln

Deduplication is important for storage efficiency, but the process of chunking and fingerprinting data is time-consuming and CPU-intensive. The presenters propose P-Dedupe, which exploits parallelism and pipelines to avoid this computation bottleneck. P-Dedupe divides the data stream into multiple sections that can be chunked and processed in parallel, with the boundaries between sections requiring special processing to account for the sliding windows used in chunking.

High-Throughput Direct Data Transfer Between PCIe SSDs

Jun Suzuki, Masato Yasuda, Masahiko Takahashi, Yoichi Hidaka, Junichi Higuchi, Yoshikazu Watanabe, and Takashi Yoshikawa, NEC Corporation

Data reallocation and backup are examples of data being transferred between devices without modification. Currently, this transfer must traverse main memory of the server hosting the I/O devices; this link can become the bottleneck. The presenters propose DirectConnect, a method to transfer this data directly, using memory in a PCIe-to-Ethernet bridge as an intermediate buffer for DMAs of the source and destination devices. Prototype evaluation shows high throughput even when server bandwidth is narrow.

Grouping Data for Faster Rebuilds: The Art of Failing Silently

Avani Wildani and Ethan Miller, University of California, Santa Cruz

In big systems with erasure coding for reliability, rebuild time after failure is inevitable and slow—up to six hours to rebuild a 300 GB disk. The goal of this work is to reduce the impact of rebuild by striping intelligently. Data is grouped into access groups that correspond to real-life working sets for applications, users, or projects. Striping these access groups strategically can ensure that a rebuild halts progress for only a few users or projects rather than all of them—one project must rebuild all of its data, rather than rebuilding some data for each of many projects. Ongoing work includes evaluation with probabilistic fault injection, modeling correlated failures, and measuring overall impact of rebuilds.

Toward an Economic Model of Long-Term Storage

Daniel C. Rosenthal, University of California, Santa Cruz; David S.H. Rosenthal, Stanford University Libraries; Ethan L. Miller and Ian F. Adams, University of California, Santa Cruz; Mark W. Storer, NetApp; Erez Zadok, Stony Brook University

People want to store their content indefinitely, but want to pay for that storage up-front rather than on a continuing basis. The cost of indefinite storage is difficult to predict, depending on future events ranging from regular disk replacement to natural disasters. The presenters use Monte Carlo modeling to simulate hypothetical futures for a storage system. They calculate tradeoffs between cost and the likelihood of data survival, in an attempt to value the endowment needed to preserve data.

Emulating a Shingled Write Disk

Rekha Pitchumani, University of California, Santa Cruz; Yulai Xie, Huazhong University of Science and Technology; Andy Hospodor, University of California, Santa Cruz; Ahmed Amer, Santa Clara University; Ethan L. Miller, University of California, Santa Cruz

Shingled disks can more than double disk capacity, but, due to their architecture, random writes may destroy data. In particular, in-place overwrites can be destructive. Research and development of how to best manage shingled disks is hindered because they are not yet available for testing. The presenters' goal is to emulate shingled disks by providing a device driver that mimics their operations. The driver uses a mapper that maintains knowledge of which tracks are overwritten by which writes, so reads to overwritten tracks return the overwritten rather than the original content. Future work includes adding the ability to report physical geometry. The emulator can be useful even after shingled disks become available as a platform for consistent and controllable testing. The emulator is currently being evaluated and will be released soon.

Cloud

Summarized by Daniel Fryer (dfryer@cs.toronto.edu)

BlueSky: A Cloud-Backed File System for the Enterprise

Michael Vrable, Stefan Savage, and Geoffrey M. Voelker, University of California, San Diego

Michael introduced BlueSky by stating that since many services are moving towards the cloud, they wanted to explore the idea of a network file server backed by cloud storage. Existing cloud storage acts like another level in the storage hierarchy, but with different characteristics. The interface usually only supports writing complete objects, but if one supports random reads, latencies are high enough that the additional penalty of random access doesn't matter. Privacy becomes more of a concern, and since billing for storage is usually by quantity used, deleting unused data is important.

Their approach to providing a cloud-backed NFS service avoids modifying the client NFS/CIFS stack. Instead, they implemented a caching proxy server. The proxy can provide lower latency, perform write-back caching, and encrypt before forwarding requests to the cloud storage service. On the back-end, they use a log-structured file system. When writing, each segment is uploaded by the proxy all at once and is stored as an object in the cloud. For reads, they take advantage of the ability to do random access on the content of these segments. There is also a garbage-collecting log cleaner process, which can run on the proxy or on a compute node in the cloud.

To maintain confidentiality, the log-cleaning process does not need to have the encryption key for the file system, which can remain safely on the proxy. They do need to make some metadata available, so they structure their metadata as a four-level tree. The top two levels are the log checkpoints and the inode map, which locate the most recent versions of inodes in the log. These levels are unencrypted. Below these are the inodes and data blocks, which do have encrypted contents. Michael then presented a diagram of the proxy architecture. At the front-end are NFS or CIFS interfaces to handle client requests. Since they do writeback caching, they write to the local disk before replying to the client to announce that a write is durable. Once they have accumulated a log segment's worth of data, they can encrypt it and use cloud-specific back-ends (S3, WAS) to store the log segment in the cloud.

Their design is predicated on high-bandwidth connections to the cloud service provider. One of the major problems is latency, which is partly a function of location. Measuring performance with varying object sizes and amounts of concurrency, they showed that 32 concurrent connections could saturate a 1 Gbps link.

To benchmark their system, they ran a kernel source unpack, checksum, and compile process. Michael compared a local NFS server, a purely remote NFS server, BlueSky with a warm cache, and BlueSky with a cold cache. They also evaluated cache hit ratios and the effect they had on client performance. With about a 50% hit rate, they were able to keep read latencies within 2x or 3x of the purely local solution. They could write at local speed until the proxy ran out of disk space for logging, at which point they were limited by bandwidth to the cloud. Michael then presented a final benchmark, based on SPECsfs2008. BlueSky performed similarly to the local NFS system with unconstrained network bandwidth; with a constrained network it scaled to about 90% of the local throughput before dropping off and becoming erratic. They

also found that while fetching full segments was helpful for the compile benchmark, it had a negative impact on SPECsfs.

Based on S3's pricing model for bandwidth and operation counts, they calculated the costs of BlueSky. The main point was that by aggregating writes into log updates and by allowing random reads, they decreased usage costs dramatically.

Brent Callaghan (Apple) asked whether they had thought about multiple proxies accessing the same data store; he also wanted to know if they had thought about backup. Michael explained that they'd thought about some of the issues but hadn't implemented any of their ideas. There are several reasons why you might want to have multiple proxies: for higher scalability or for geographically distributed access. One approach would be to have multiple proxies writing to separate logs in the cloud and rely on some kind of opportunistic concurrency, or maybe implement distributed logging. For backups, as long as you don't garbage-collect all your log segments, you can get information from a previous checkpoint.

Someone from Nimble Storage wondered whether log structuring was worth it, given the complications of cleaning. Why couldn't you just increase throughput with higher concurrency? Michael explained that the major reason is cost; you pay an operation cost on a per-object basis. The cleaning can run in the cloud, so you don't pay transfer charges.

Someone from Red Hat asked what their benchmark NFS server was, because the numbers looked horrible. They were referred to the numbers in the paper, but it was a Linux server with a couple of disks. The numbers might have been different had they used a high-performance storage server. Joe Tucek of HP Labs asked whether they had thought about the different consistency models provided by the different cloud services. Michael replied that, because BlueSky is log-structured, they're not overwriting data in place, so eventual consistency doesn't cause them as much trouble. They don't have different versions of objects; they're either there or not there. If something just isn't there, they could retry, timeout, or report an error to the client.

Rethinking Erasure Codes for Cloud File Systems: Minimizing I/O for Recovery and Degraded Reads

Osama Khan and Randal Burns, Johns Hopkins University; James Plank and William Pierce, University of Tennessee; Cheng Huang, Microsoft Research

Osama Khan discussed the rapid growth in the total quantity of stored data, projecting a 44-fold growth over 10 years, particularly in the cloud. With this much data, replication is not a cost-effective means to achieve reliability. Erasure coding is a natural solution to this problem, but with traditional erasure-coding approaches, recovery is a slow process requiring the involvement of many devices. Existing erasure codes are not designed with minimal recovery I/O in mind.

Their solution is to create an algorithm that minimizes the amount of data needed for recovery under any XOR-based erasure code. Before describing the details, Osama presented an overview of the general process of erasure coding, starting with a block of file system data. The encoding is done by a matrix multiplication, and then the result is distributed into stripes after encoding. He gave an example of the type of decoding equation that results from this process. Their algorithm finds a decoding equation for each failed bit while minimizing the total number of symbols needed for reconstruction, given the code generator matrix and a list of failed symbols. They do this by constructing a directed graph, with the weights on the edges representing the number of symbols involved in the equation. The lowest-cost path through the graph minimizes the number of symbols involved. These solutions can be precomputed and stored for later use.

Their second contribution was to address the problem of degraded reads—disks that are temporarily unable to deliver data. To optimize read performance to deal with this, they invented a new class of codes called "rotated Reed-Solomon codes." Standard coding schemes compute different symbols from single rows, whereas rotated codes span multiple rows. This means that each coding disk is using slightly different symbol sets. Osama presented some examples of what kind of access has to be done during failure and how the rotated Reed-Solomon codes require fewer reads.

Jay Wiley asked how their graph-based algorithm compared to Hafner's work using matrix methods. Osama couldn't remember, so they took it offline. If their symbol size was 100—500 MB for performance reasons, Jim Molina of Western Digital wondered, what kind of correction capacity would they have? It was clarified that the block size doesn't affect correction capacity, which is a function of redundancy.

NCCloud: Applying Network Coding for the Storage Repair in a Cloud-of-Clouds

Yuchong Hu, Henry C.H. Chen, and Patrick P.C. Lee, The Chinese University of Hong Kong; Yang Tang, Columbia University

Patrick noted how outages or vendor lock-in makes depending on a single cloud provider for storage risky. The obvious solution is to take advantage of multiple-cloud storage, using a proxy to stripe data across the clouds using an MDS encoding scheme where any K out of N clouds can reconstruct the original data. Repair would then involve downloading all the data from the functioning clouds to determine what to write to a new cloud. This could incur a high repair cost due to bandwidth usage equivalent to the size of the whole dataset.

Their system, NCCloud, applies the idea of "regenerating codes" to the problem of repair in bandwidth-constrained situations. Regenerating codes aim to reduce the amount of data needed to perform reconstruction of a failed node by selectively downloading portions of the data stored on each node, where the nodes themselves may perform some computation on the data during the reconstruction process. Up to this point, regenerating codes have primarily been studied from a theoretical perspective. To keep NCCloud simple, they would like to avoid any computation on the storage nodes.

NCCloud relies on their implementation of a functional minimum-storage regenerating code (F-MSR). Reconstruction is based on random linear combinations of existing chunks. Unlike a "systematic" code, they don't keep the original data around, but only the linearly combined code chunks. This makes actual decoding expensive. They propose F-MSR for rarely read long-term archival applications. One challenge that arises is ensuring that after reconstruction, the MDS properties of the original encoding are preserved and that any subsequent repair will preserve properties as well. F-MSR reduces repair bandwidth cost by 25%. They compare NCCloud with F-MSR to Reed-Solomon–based RAID-6. F-MSR has higher response time during writes, due to encoding overhead, which they expect will be masked by network latency unless N is very large. Reconstruction time is lower, due to less bandwidth use. In summary, NCCloud realizes an implementation of a regenerating code, which preserves storage cost but uses less repair traffic.

Someone asked what the odds were of losing data from a single cloud provider, much less two. Patrick argued that there are many new cloud providers, and we can't guarantee that they are all equally reliable or available. The questioner said that he thought that the math was really interesting, but he didn't think the economics of it made sense.

Poster Session II

See static.usenix.org/events/fast12/poster.html for PDFs and descriptions of all posters.

A Little Bit of Everything

Summarized by Doowon Kim (dwkim@cs.utah.edu)

Extracting Flexible, Replayable Models from Large Block Traces

V. Tarasov and S. Kumar, Stony Brook University; J. Ma, Harvey Mudd College; D. Hildebrand and A. Povzner, IBM Almaden Research; G. Kuenning, Harvey Mudd College; E. Zadok, Stony Brook University

Vasily Tarasov described what their traces look like. In general, a timestamp is a common field, but other fields depend on what events are traced. The authors used block traces focusing on operation, I/O size, and offset, but their approach is valid for any trace. Vasily said that there are two main use cases for traces: (1) workload analysis and characterization and (2) trace replay. He mentioned that trace replay has some problems.

Vasily showed why statistics matter. Although traces collected on the same machine and in the same environment might differ on Monday and Tuesday, for example, it's the overall statistical modeling of properties such as I/O rate and read-write ratio that are important to consider in evaluating systems.

He explained their design goals: (1) accuracy, (2) conciseness, (3) flexibility, and (4) extensibility. Vasily explained that the first problem they encountered was that workload can change in the trace over time. He explained the feature functions within a chunk and said that they can put the value into a multi-dimensional histogram. He then said they could generate benchmark plugins and explained what the plugins are. In the evaluation of their work, he argued that the average relative error is less than 10% across all parameters and systems, and there was a 17x–25x size reduction. Vasily then discussed their future work: (1) more accurate parameters, systems, and traces; (2) file system traces; (3) automatic selection of parameters; and (4) operations on models.

Joe Tucek from HP Labs asked about the difference between Monday and Tuesday traces. Vasily replied that many assume the traces will be the same day-to-day, but there may be significant differences in the pattern of use, while the overall load remains the same. They want to develop meta-cases and be able to work from that. Someone from VMware asked about the chunk sizes used in deduplication. Vasily said that he understood the issue, but that they didn't include the information about chunk sizes in the paper. Someone from Microsoft asked if it takes an expert to choose from their library of functions that can create particular traces. Vasily said they had published a tool named Distiller that helps with this with good results. Josh Berry of Riverbed Technology asked if they had looked at latency-dependent workloads. Vasily said that this is a known problem with traces, and they did experiment with adding in random delays or having no delays (infinite speed).

scc: Cluster Storage Provisioning Informed by Application Characteristics and SLAs

Harsha V. Madhyastha, University of California, Riverside; John C. McCullough, George Porter, Rishi Kapoor, Stefan Savage, Alex C. Snoeren, and Amin Vahdat, University of California, San Diego

John McCullough started by explaining provisioning hardware for cluster applications. There are many goals for provisioning, but he focused only on achieving SLA (perfor-

mance) goals and minimizing cost for a single application while emphasizing storage. He said that the challenge is a very large configuration space, making solving this problem non-trivial. The current state-of-the-art solution is just to apply rules-of-thumb from experience, and use trial-and-error with various configurations. Their goal is to discover what a low-cost configuration is now and what a low-cost configuration will look like in the future. They do this by first measuring "in-the-small" and modeling application performance, in order to predict "in-the-large." He explained scc (Storage Configuration Compiler). If cluster building blocks, an application model, and SLA specification are put into scc, it produces a spectrum of cost for different configurations.

John explained cluster building blocks. Servers have many components, such as cores, RAM, storage, I/O, and network. You also need an application model to use scc. The model has tasks (computation), datasets (storage), edges between tasks and datasets (I/O), and edges among tasks (dependencies). In his example of the model, photo-sharing, there are three datasets: photos, thumbnails, and tags. The related tasks are single operations with known sizes for writing or reading from the datasets. John said that if you use only a hard disk and single core, the cost will be really low. However, if you use a lot of SSDs, the cost will be high. The guiding principle is to meet SLA. He said ILP minimizes cost.

John talked about validation. They built three applications: photo-sharing, product search, and Terasort. He showed that scc meets the SLA at lower cost for all three.

Sanghyun Cho from University of Pittsburgh asked whether the author had considered including costs such as power. John answered that currently they did not include power bills. Fred Douglis of EMC asked whether they can handle very large models and had tried perturbing the inputs. John answered that for larger models they use a gradient descent to pick the best solution. In terms of perturbing, some of that can be done by swapping out parts. Ben Reed of Yahoo! said that this work reminded him of the Starfish project in the database community, as they had both a processing and a working-set model built by profiling. John said he wasn't familiar with Starfish, but they did get feedback from unnamed storage providers. Ben said that just tuning the software configuration made huge differences. John said that this sounds interesting to pursue, but complex. Someone from Google wondered about using scc for a cluster that would have multiple, simultaneous uses. John said they planned to look into that in the future. Randal Burns (Johns Hopkins) pointed out that this is not the way people deploy in the cloud. John said they want to extend their model to support cloud deployments. Rik Farrow asked if scc is available for use, and John replied that he would have to ask his co-authors.

iDedup: Latency-aware, Inline Data Deduplication for Primary Storage

Kiran Srinivasan, Tim Bisson, Garth Goodson, and Kaladhar Voruganti, NetApp, Inc.

Kiran Srinivasan began by explaining the overview and context of iDedup. He said that storage clients are connected to primary storage via NFS, CIFS, or iSCSI, and the primary storage is connected to secondary storage through NDMP or other methods. In this hierarchy, dedupe can save more than 90% in secondary storage. However, primary storage has some unique characteristics. First, performance and reliability are key features. Second, RPC-based protocols are very latency sensitive. Third, only offline dedupe techniques have been developed. He said iDedup is for inline or foreground dedupe for primary storage and has little impact on latency-sensitive workloads. Kiran compared offline dedupe to inline dedupe. He explained why inline dedupe for primary storage is required. He said that this is because provisioning and planning is easier, with no post-processing activities, and allows efficient use of resources. He then explained the key features of iDedup. First, it minimizes inline dedupe performance overheads. Second, it has a tunable tradeoff. Last, it can be combined with offline techniques.

Kiran talked about inline dedupe challenges. First, it has read path challenges. This is because dedupe causes disk-level fragmentation. Second, it has write path challenges, because it produces CPU overheads in the critical write path and extra random I/Os in the write path due to the dedupe algorithm. He then talked about their approach: iDedup provides a solution to read path issues which are dedupe-only sequences of disk blocks, as well as keeping a smaller dedupe metadata as an in-memory cache to write path issues.

Kiran explained that the iDedup architecture has two design-tunable parameters: threshold and dedupe metadata (fingerprint DB) cache size. Kiran then explained the iDedup algorithm and its four phases.

In their evaluation setup, they replayed real-world CIFS traces. Kiran compared iDedup to a system with no iDedup and with full dedupe. They tested three dedupe metadata cache sizes: 0.25, 0.5 and 1 GB. Results showed less than a linear decrease in dedupe saving, and that the ideal threshold is the biggest threshold with the least decrease in dedupe saving. Fragmentation for other thresholds is between the baseline and threshold 1. CPU utilization demonstrated a larger variance compared to the baseline, but the mean difference was less than 4%. Finally, Kiran showed that the result of latency impact for longer response times is larger than 2 ms.

Margo Seltzer asked about the result graph (in Figure 7) comparing the deduplication ratio to the minimum sequence threshold. Kiran answered that they chose to use 4 as the

best threshold. Margo then asked about prior work done on inline deduplication at UCB. Kiran answered that this work was similar, but completely redone. Joseph Glider of IBM Almaden Research asked whether their traces include content information. Kiran said that they used content hashes. Glider then asked whether they consider the age of an entry before ejecting it from the cache. Kiran answered that they do not use LRU. The policy they use is based on hashes for blocks that have been previously used for deduplication. Vasily Tarasov (Stony Brook) asked if they preserve the dedup information when moving from primary storage to secondary. Kiran answered that they didn't but it was a good idea.

Flash and SSDs, Part II

Summarized by Doowon Kim (dwkim@cs.utah.edu)

Caching Less for Better Performance: Balancing Cache Size and Update Cost of Flash Memory Cache in Hybrid Storage Systems

Yongseok Oh, University of Seoul; Jongmoo Choi, Dankook University; Donghee Lee, University of Seoul; Sam H. Noh, Hongik University

Yongseok Oh explained that hybrid storage systems benefit from combining SSDs and HDDs. One of the important characteristics of flash-based SSD is that it maintains over-provisioned space (OPS). Typical SSDs have a fixed OPS size in which optimal size is unknown, so one of their goals was to determine the optimal size of OPS. According to Oh, as OPS increases, the performance cost of garbage collection (GC) decreases but the cache miss rate increases. Overall, the performance is going to be bad, but optimal OPS size can produce the best performance possible.

Oh presented various cost models and then moved into an explanation of his evaluation setup. He used a hybrid storage simulator and flash cache layers (FCLs). OP_FCL shows near-optimal performance, and optimal performance depends on workload characteristics. OP-FCL dynamically adjusts cache spaces according to workloads. Considerable OPS is used to lower garbage collection cost. Most caching space is used to maintain read data. Optimizing the lifetime of the flash is left as future work.

Umesh Maheshwari of Nimble Storage said the paper was very interesting and he thought it is very useful in real SSD developing. Then he asked whether read cost's dependence on garbage collection cost was their assumption or their experience. Oh answered (with his advisor's help) that in reality, the read should not be affected by garbage collection, but that their assumption was valid because they experienced that. Someone from Google wondered how a cache policy like LRU or FIFO affects the cost model. Oh answered that they did not look at other replacement policies. In this case they used LRU, but, overall, the cost model will not be changed.

Lifetime Management of Flash-Based SSDs Using Recovery-Aware Dynamic Throttling

Sungjin Lee and Taejin Kim, Seoul National University; Kyungho Kim, Samsung Electronics, Korea; Jihong Kim, Seoul National University

Lee said that flash-based SSDs are becoming an attractive storage solution for enterprise systems, but poor write endurance results in the limited lifetime of SSDs hampers wideradoption of SDDs. The SSD lifetime is determined by the number of bytes that can be written to the SSD and the number of bytes written per day. Mobile phones and desktop PCs that are not write-intensive can achieve the required lifetime, but with write-intensive workloads such as on enterprise servers, a reasonable lifetime cannot be guaranteed. Flash has a self-healing capability that increases flash lifetime related to the logarithm of the time between erasures. Current schemes such as reducing WAF and incoming write traffic can improve overall SSD lifetime but cannot guarantee the required SSD lifetime. Static throttling limits the maximum throughput of SSDs but is also likely to throttle performance uselessly and to underutilize the available endurance.

Lee introduced Recovery-Aware Dynamic Throttling (READY). READY's design goals are to guarantee the required SSD lifetime, minimize average response times, and minimize response time variations. READY consists of three modules: the write demand predictor, the throttling delay estimator, and the epoch-capacity regulator. The write demand predictor can predict future write traffic for throttling by exploiting cyclic behaviors of enterprise applications. Throttling delay should match future write demand, increasing if demand exceeds epoch capacity, decreasing if demand falls short of capacity, and remaining unchanged if demand equals capacity. The epoch-capacity regulator can throttle write performance by applying the same throttling delay to every page write and increasing a throttling delay later to reclaim the overused capacity.

The team evaluated four SSD configurations: NT, ST, DT, and READY. Results showed that NT cannot guarantee the required SSD lifetime, READY achieves a lifetime close to five years, and ST and DT exhibit a lifetime much longer than five years. NT exhibited the best performance, and READY performed better than ST and DT while guaranteeing the required lifetime. Finally, READY showed shorter response time variations than ST/DT, and ST exhibited the most significant response time variations. Future work involves implementing READY in a real SSD platform and supporting latency-aware performance throttling.

usenix
THE ADVANCED
COMPUTING SYSTEMS
ASSOCIATION

USENIX Association
2560 Ninth Street, Suite 215
Berkeley, CA 94710

POSTMASTER
Send Address Changes to *;login:*
2560 Ninth Street, Suite 215
Berkeley, CA 94710

PERIODICALS POSTAGE
PAID
AT BERKELEY, CALIFORNIA
AND ADDITIONAL OFFICES