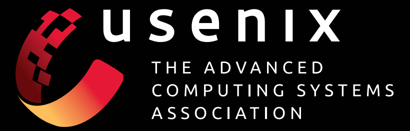
;login:

OCTOBER 2013 VOL. 38, NO. 5



sysadmin

On Leadership
Tom Limoncelli

Synnefo, Cloud Management System
Vangelis Koukis, Constantinos Venetsanopoulos, and Nectarios Koziris

Log Filtering with Rsyslog
David Lang

Clientside SSD and Network Storage
David Holland, Elaine Angelino, Gideon Wald, and Margo Seltzer

Columns

Practical Perl Tools: Domain-specific Parsers
David Blank-Edelman

Python: Using Context Managers
David Beazley

Hearsay Among Monitoring Systems
Dave Josephsen

For Good Measure: Annual Cyber Security Report
Dan Geer and Mukul Pareek

/dev/random: Job Hunting
Robert Ferrell

Conference Reports

HotOS XIV: 14th Workshop on Hot Topics in Operating Systems

HotPar ’13: 5th USENIX Workshop on Hot Topics in Parallelism

usenix THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

UPCOMING EVENTS

LISA '13: 27th Large Installation System Administration Conference
November 3–8, 2013, Washington, D.C., USA
www.usenix.org/conference/lisa13

SESA '13: 2013 USENIX Summit for Educators in System Administration
CO-LOCATED WITH LISA '13
November 5, 2013, Washington, D.C., USA
www.usenix.org/conference/sesa13

FAST '14: 12th USENIX Conference on File and Storage Technologies
February 17–20, 2014, Santa Clara, CA, USA
www.usenix.org/conference/fast14

2014 USENIX Research in Linux File and Storage Technologies Summit
IN CONJUNCTION WITH FAST '14
February 20, 2014, Mountain View, CA, USA
www.usenix.org/conference/linuxfastsummit14
Submissions due: January 17, 2014

NSDI '14: 11th USENIX Symposium on Network Systems Design and Implementation
April 2–4, 2014, Seattle, WA, USA
www.usenix.org/conference/nsdi14

2014 USENIX Federated Conferences Week
June 17–20, 2014, Philadelphia, PA, USA

USENIX ATC '14: 2014 USENIX Annual Technical Conference

HotCloud '14: 6th USENIX Workshop on Hot Topics in Cloud Computing

WiAC '14: 2014 USENIX Women in Advanced Computing Summit

HotStorage '14: 6th USENIX Workshop on Hot Topics in Storage and File Systems

UCMS '14: 2014 USENIX Configuration Management Summit

ICAC '14: 11th International Conference on Autonomic Computing

USENIX Security '14: 23rd USENIX Security Symposium
August 20–22, 2014, San Diego, CA, USA
Submissions due: February 27, 2014

EVT/WOTE '14: 2014 Electronic Voting Technology Workshop/Workshop on Trustworthy Elections

USENIX Journal of Election Technology and Systems (JETS)
Published in conjunction with EVT/WOTE
www.usenix.org/jets
Submissions for Volume 2, Issue 2, due: December 5, 2013
Submissions for Volume 2, Issue 3, due: April 8, 2014

HotSec '14: 2014 USENIX Summit on Hot Topics in Security

FOCI '14: 4th USENIX Workshop on Free and Open Communications on the Internet

HealthTech '14: 2014 USENIX Workshop on Health Information Technologies
Safety, Security, Privacy, and Interoperability of Health Information Technologies

CSET '14: 7th Workshop on Cyber Security Experimentation and Test

WOOT '14: 8th USENIX Workshop on Offensive Technologies

OSDI '14: 11th USENIX Symposium on Operating Systems Design and Implementation
October 6–8, 2014, Broomfield, CO, USA

Diversity '14: 2014 Workshop on Diversity in Systems Research
CO-LOCATED WITH OSDI '14

LISA '14: 28th Large Installation System Administration Conference
November 9–14, 2014, Seattle, WA, USA

Stay Connected...

twitter.com/usenix
www.usenix.org/facebook
www.usenix.org/youtube
www.usenix.org/linkedin
www.usenix.org/gplus
www.usenix.org/blog

;login:

OCTOBER 2013 VOL. 38, NO. 5

EDITORIAL

2 Musings *Rik Farrow*

CLOUD

6 Synnefo: A Complete Cloud Stack over Ganeti
Vangelis Koukis, Constantinos Venetsanopoulos, and Nectarios Koziris

11 vPipe: One Pipe to Connect Them All
Sahan Gamage, Ramana Kompella, and Dongyan Xu

16 Hyper-Switch: A Scalable Software Virtual Switching Architecture
Kaushik Kumar Ram, Alan L. Cox, and Scott Rixner

SYSADMIN

20 Technical Leadership Is Something We Can All Do
Tom Limoncelli

23 Log Filtering with Rsyslog
David Lang

30 Flash Caching on the Storage Client
David A. Holland, Elaine Angelino, Gideon Wald, and Margo I. Seltzer

PROGRAMMING

36 Valerie Aurora on File Systems and the Ada Initiative
Rikki Endsley and Valerie Aurora

40 Modular SDN Programming with Pyretic
Joshua Reich, Christopher Monsanto, Nate Foster, Jennifer Rexford, and David Walker

48 Drilling Network Stacks with packetdrill
Neal Cardwell and Barath Raghavan

COLUMNS

54 Practical Perl Tools: Parse Me, Amadeus *David N. Blank-Edelman*

59 Python: With That: Five Easy Context Managers *David Beazley*

64 iVoyeur: Hearsay *Dave Josephsen*

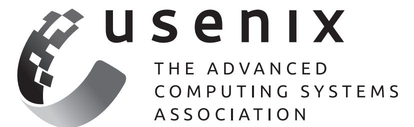
68 For Good Measure: Trending North *Dan Geer and Mukul Pareek*

72 /dev/random *Robert G. Ferrell*

74 Book Reviews *Elizabeth Zwicky, Mark Lamourine, and Melissa Gray*

CONFERENCE REPORTS

80 14th Workshop on Hot Topics in Operating Systems (HotOS XIV), 5th USENIX Workshop on Hot Topics in Parallelism (HotPar '13)



EDITOR
Rik Farrow
rik@usenix.org

MANAGING EDITOR
Rikki Endsley
rikki@usenix.org

COPY EDITOR
Steve Gilmartin
proofshop@usenix.org

PRODUCTION
Arnold Gatilao
Casey Henderson
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
www.usenix.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 non-members are $90 per year. Periodicals postage paid at Berkeley, CA, and additional offices.

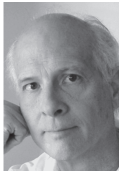
POSTMASTER: Send address changes to *;login:*, USENIX Association, 2560 Ninth Street, Suite 215, Berkeley, CA 94710.

©2013 USENIX Association
USENIX is a registered trademark of the USENIX Association. Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. USENIX acknowledges all trademarks herein. Where those designations appear in this publication and USENIX is aware of a trademark claim, the designations have been printed in caps or initial caps.

EDITORIAL

Musings

RIK FARROW



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

Like last issue, I can still see clouds on the horizon. But this time the clouds are white and puffy, not the dark stormy ones of August. These clouds are elastic, growing in the afternoons and shrinking at night. Sort of like, well, the clusters of VMs we call clouds.

If we go back to the decade of the nineties, cluster computing was the big thing: you could build a Beowulf cluster and "have a supercomputer in your den" [1]. People are still building Beowulf clusters today, but instead of building supercomputers, some are using Raspberry Pi's and lots of blinking lights [2]. Beowulf clusters required identical computers, running open source software including libraries for parallel processing, like MPI.

By the end of the nineties, something else was happening. A small company had built a cluster designed with one purpose in mind: checking backlinks to determine page rank. Google needed to search billions of Web pages quickly and cheaply so that they could return search results quickly. Other companies also began building clusters, and these clusters, like the mainframes that came before them, ran two types of jobs: interactive services and batch jobs.

Taking a parallel path through history, VMware also began in the late nineties, and was selling a hypervisor by 2001. While virtual machines had started out as a way of running multiple copies of single-user operating systems (CP/CMS) on expensive mainframes [3], the clouds we talk about today are ones that run VMs on top of hypervisors on clusters of computers.

Elastic Clouds

And that's where the fluffiness of clouds that I started off with comes in. I was listening to Eric Brewer's keynote at HotPar '13 [4] as he explained some key issues with clusters and clouds. Brewer pointed out that latency matters a lot, and making potential customers wait even a few hundred milliseconds was bad for business. As companies like Google and Amazon realized this, they focused on improving the customer experience through tiered, parallel systems, and caching. As parallelism increases, so does latency, where the slowest response to a client request results in the entire response appearing slow. At this point, Brewer suggested that if you want to know whether a service is running on virtualized servers, just measure tail latency, a measure of the number of requests that fall beyond 99% of the desired latency window. I'll have more to say about this later.

Brewer then explained that with public facing services, peak demand can be six or ten times as much as average demand. Because you can't let your customers wait, even for an extra half second, you need to allocate resources for servicing that peak demand. And that suggests that you will be idling 84–90% of your servers most of the time. The way beyond this wasteful state leads us to clouds.

By being able to sell or use compute servers off-peak, you can soak up that idle time. For companies like Amazon, you offer spot prices which can be 10% of on-demand prices for compute servers. If you are Google (or eBay, Yahoo!, and dozens of other companies), you use your non-peak resources to run batch jobs. Either way, your goal is to get the maximum utilization out of your servers, all the time.

For internal clusters, you not only control the load from batch jobs, you also control the software that is running on your cluster. For compute services that are publicly offered, you have some control over the load, and little over the software that is being run (the main exception being noticing and killing off VMs that are scanning, spamming, and DoSing the Internet). As a prophylactic, publicly available servers all rely on virtual machines, as they provide a degree of isolation between the host and the software that someone has loaded and is running on it.

Virtually True

In many ways, clouds based on the ability to run virtual machines are a wonderful invention. They allow sharing of precious resources that otherwise would be idle. They support the use of familiar interfaces, so programmers aren't faced with an unfamiliar environment. And they do provide some real isolation between the hosted OS and the host and the rest of the cloud. But there's the rub.

We can't have all the wonderful things that clouds provide without paying a price. And that price comes in terms of both performance and latency. Applications in VMs must cross protection rings twice, once from the hosted VM into the VMM, then again when accessing devices in any driver domain (noting that not all VMMs require this). Another performance cost comes from sharing resources: if a VM is not scheduled when a network packet or disk block arrives, it must wait. This is a large source of the tail latency that Brewer was speaking about, which I said I'd get to later. And, finally, the use of VMs means that the underlying VMM loses information that operating systems have traditionally used to improve performance, such as disk block caching.

Brewer went on to describe a research operating system, Akaros [5], that is "made for the cloud." Akaros supports both provisioning and allocation of resources. Services that require low latency guarantees get provisioned, which means they can always have enough resources, even during peak load. Other applications receive allocations that can be revoked within two or three microseconds. Akaros supports Linux libc, so it does provide a familiar programming environment. And they are designing Akaros so clouds are no longer necessary for many jobs. In the Akaros model, each application runs on bare metal. But the isolation properties of VMs are not part of Akaros, so clouds will not be banished with this design for public-facing compute servers.

Techniques for improving the performance of VM I/O have been the topic of many papers, and several articles in this magazine as well. I have long had the intuition that virtualization was neither the best performing nor most secure path we could take—even if it was an easier one than starting over with other OS models designed with principals like Akaros' provisioning and allocation, as well as the isolation that it currently lacks.

The Lineup

We start off this issue with an article about Synnefo. Synnefo is a cloud stack that runs on top of Ganeti clusters, and is also open source software, sporting both command-line and attractive GUI interfaces, and has been in use for years in Greece. The three lead developers of Synnefo, Vangelis Koukis, Constantinos Venetsanopoulos, and Nectarios Koziris, have written a wonderfully clear description of the parts that form Synnefo, as well as explaining how it fits in with Ganeti [6] and OpenStack.

Next, we have an article by Gamage et al. about a technique they have been working on to improve VM I/O. vPipe fixes some of the issues I just wrote about by passing more information to the VMM (Xen in this case) so that a VM can hand off some of the tasks that are more efficiently done within the VMM, such as copying a file to a network socket.

Ram et al. have developed Hyper-Switch, a virtualized switch designed to improve network switching performance. Also working in Xen, Hyper-Switch moves the control plane out of the device domain and into the hypervisor, which knows which VMs are currently scheduled, and can wake up VMs when it makes the most sense to do so, showing a large improvement in switching performance.

Tom Limoncelli has written a relevant (and short) article about leadership. At first, I wondered just where he was going. But once I got the point, I was sure I would remember his point about leadership, and try to practice it.

David Lang has written about ryslog. David is both a committer and a user of rsyslog, which he mentioned in his first article about enterprise logging in the June 2013 issue. In this article, David provides examples of the many ways that rsyslog can filter, modify, and even store log messages.

David Holland et al. decided to take a careful look at whether the client-side flash cache helps servers that use network file servers. The good news is that for applications with a large working set, a flash cache can help a lot. While a type of client-side flash cache is already available from NetApp, it's something we may be seeing more of in the future.

Rikki Endsley, the USENIX Community Manager, interviewed Val Aurora. Aurora had been a Linux kernel committer, with a focus on file systems. More recently, she has started the Ada Initiative, a nonprofit dedicated to promoting women in open tech/culture.

Josh Reich et al. have written about Pyretic, software for creating policies for OpenFlow hardware. OpenFlow has become increasingly important for datacenter networks, but OpenFlow's programming interface, according to these authors, is closer to assembler than an API. Pyretic allows you to compose policies and apply them to multiple OpenFlow devices, and this article explains both OpenFlow and Pyretic.

Neal Cardwell and Barath Raghavan wrote about packetdrill, a tool for troubleshooting network protocols and stacks. Although most of us will not be writing network stacks, packetdrill seems like a tool that may be useful to anyone who is having problems with a networked application and wondering exactly what went wrong where.

David Blank-Edelman decided to explore domain-specific languages through the use of Perl parsing modules. Although you may not be planning on making recipe parser, understanding the tools for parsing your own language may help you someday.

David Beazley explores context managers in Python. For example, you can have files closed automatically or locks acquired and released using context managers, and David explains how this works and how to create your own.

Dave Josephsen decided to scratch an itch: how to get different monitoring systems talking to each other. Well, not so much talking to each other—Nagios, Collectd, Ganglia, and Splunk can do that already—as talking via a centralized, simplified service. Dave is building that service, called Hearsay, and wants your input and help.

Dan Geer and Mukul Pareek share their Index of Cyber Security findings. The ICS is based on monthly surveys of professionals who work in large organizations and with security, and the goal of ICS is to provide current trends in security. Some things do change, such as which component risks are higher each month, while other things stay the same.

Robert Ferrell takes us on a strange adventure: job hunting. Well, many people wouldn't find job hunting quite as unusual or interesting as Robert does. And if you've ever wanted to write the perfect job description for that best system administrator you'll never be able to hire, you need to read his column.

Elizabeth Zwicky has reviewed four books this month. She starts off with *Peopleware*, an old favorite of hers that has been revised, successfully. Elizabeth then read *Adaptive Software Development*, a book that never mentions "Agile" yet does discuss development for rapidly changing environments. Elizabeth next covers *The Practice of Network Security Monitoring* by Richard Bejtlich, who is certainly an old pro when it comes to monitoring. She ends with *Graph Databases*, a book that explains how databases can cover objects and relationships—for example, Facebook friends or LinkedIn people.

Mark Lamourine kept very busy this time, starting off with a review of Stevens and Rago's updated *Advanced Programming in the UNIX Environment* (third edition). Rago has added more operating systems (FreeBSD, Linux, MacOS, and Solaris 10), while the style remains classic Stevens, thorough and comprehensible. Mark continues with *The Go Programming Language Phrasebook*, a book that provides examples of Go features, which means going pretty deep into details such as structures in memory. Finally, Mark read *The Realm of Racket*, and wondered whether the game-centric approach would really help someone understand a language based on Scheme, but he does have some help from an intern, Melissa Gray, who provided her perspective.

The October issue also includes summaries for six workshops as well as the Annual Technical Conference. These summaries will all appear online, and we will print as many as we can while staying within our environmentally conscious page limit.

Since I began writing, those fluffy, white clouds have become gray and even a bit stormy. Perhaps I need to be more careful about what I have to say about clouds. But I really do believe that while cloud technology is going to be with us for a long time, there are other alternatives that we should be researching and learning about.

References

[1] D. Spector, "Building Your Own Beowulf Cluster": http://www.wired.com/wired/archive/8.12/beowulf.html.

[2] M. Szczys, "33 Node Beowulf Cluster Built with Raspberry Pi": http://hackaday.com/2013/05/21/33-node-beowulf-cluster-built-with-raspberry-pi/.

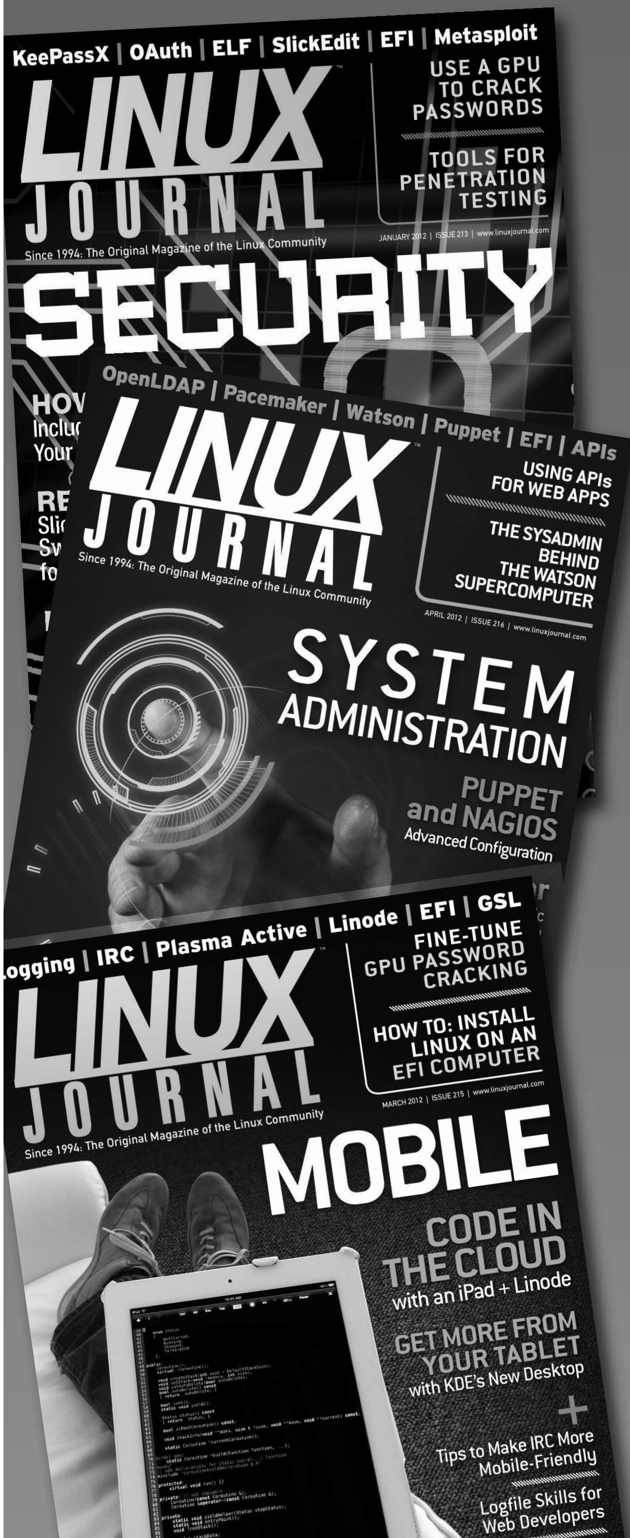
[3] CP/CMS: http://en.wikipedia.org/wiki/CP/CMS.

[4] E. Brewer, "Parallelism in the Cloud": https://www.usenix.org/conference/hotpar13/parallelism-in-the-cloud.

[5] Akaros, a research OS made for the cloud: http://akaros.cs.berkeley.edu.

[6] G. Trotter and T. Limoncelli, "Ganeti: Cluster Virtualization Manager," *;login:*, vol. 38, no. 3 (June 2013): https://www.usenix.org/publications/login/june-2013-volume-38-number-3/ganeti-cluster-virtualization-manager.

If You Use Linux, You Should Be Reading LINUX JOURNAL™



- In-depth information providing a full 360-degree look at featured topics relating to Linux
- Tools, tips and tricks you will use today as well as relevant information for the future
- Advice and inspiration for getting the most out of your Linux system
- Instructional how-tos will save you time and money

Subscribe now for instant access! For only $29.50 per year—less than $2.50 per issue—you'll have access to *Linux Journal* each month as a PDF, in ePub & Kindle formats, on-line and through our Android & iOS apps. Wherever you go, *Linux Journal* goes with you.

SUBSCRIBE NOW AT:

WWW.LINUXJOURNAL.COM/SUBSCRIBE

Synnefo: A Complete Cloud Stack over Ganeti

VANGELIS KOUKIS, CONSTANTINOS VENETSANOPOULOS, AND NECTARIOS KOZIRIS

Vangelis Koukis is the technical lead of the ~okeanos project at the Greek Research and Technology Network (GRNET). His research interests include large-scale computation in the cloud, high-performance cluster interconnects, and shared block-level storage. Koukis has a Ph.D. in Electrical and Computer Engineering from the National Technical University of Athens. vkoukis@grnet.gr

Constantinos Venetsanopoulos is a cloud engineer at the Greek Research and Technology Network. His research interests include distributed storage in virtualized environments and large-scale virtualization management. Venetsanopoulos has a diploma in Electrical and Computer Engineering from the National Technical University of Athens. cven@grnet.gr

Nectarios Koziris is a Professor in the Computing Systems Laboratory at the National Technical University of Athens. His research interests include parallel architectures, interaction between compilers, OSes and architectures, OS virtualization, large-scale computer and storage systems, cloud infrastructures, distributed systems and algorithms, and distributed data management. Koziris has a Ph.D. in Electrical and Computer Engineering from the National Technical University of Athens. nkoziris@cslab.ece.ntua.gr

Synnefo is a complete open source cloud stack that provides Compute, Network, Image, Volume and Storage services, similar to the ones offered by AWS. Synnefo manages multiple Ganeti [2] clusters at the backend for the handling of low-level VM operations. Essentially, it provides the necessary layer around Ganeti to implement the functionality of a complete cloud stack. This approach enforces clear separation between the cluster management layer and the cloud layer, a distinction that is central to Synnefo's design. This separation allows for easier upgrades without impacting VM stability, improves scalability, and simplifies administration. To boost third-party compatibility, Synnefo exposes the OpenStack APIs to users. We have developed two stand-alone clients for its APIs: a rich Web UI and a command-line client.

In this article, we describe Synnefo's overall architecture, its interaction with Ganeti, and the benefits of decoupling the cloud from the cluster layer. We focus on Synnefo's handling of files, images, and VM volumes in an integrated way and discuss advantages when choosing Synnefo to deliver a private or public cloud. We conclude with our experiences from running a real-world production deployment on Synnefo.

Layers

Before describing Synnefo itself in more detail, we will talk about the five distinct layers we recognize in building a complete cloud stack, from the lowest level, closest to the machine, to the highest level, closest to the user:

The *VM-hypervisor* layer is a single VM as created by the hypervisor. The *node* layer represents a number of VMs running on a single physical host. The software on this layer manages the hypervisor on a single physical node and the storage and network visible by the node and sets them up accordingly for each VM. The *cluster* layer is responsible for managing a number of physical nodes, with similar hardware configuration, all managed as a group. The software on this layer coordinates the addition/removal of physical nodes, allows for balanced allocation of virtual resources, and handles live VM migration. The *cloud* layer manages a number of clusters and also brings the user into the picture. The software on this layer handles authentication, resource sharing, ACLs, tokens, accounting, and billing. It also implements one or more APIs and decides how to forward user requests to potentially multiple clusters underneath. The *API* layer is not an actual software layer but rather is the API specification that should be used by the clients of the cloud platform. Finally, at the highest level, we have the *UI* layer that speaks to the platform's APIs.

Building a cloud stack is a difficult engineering problem because it spans many distinct domains. The task is complicated because it involves two distinct mindsets that meet at the cloud⟷cluster boundary. On one side is traditional cluster management: low-level virtualization and OS concepts, processes, synchronization, locking, scheduling, block storage management, network switches/routers, and knowledge that there is physical hardware involved, which fails frequently. On the other side lies the fast-paced world of Web-based development, Web services, rich UIs, HTTP APIs, REST, JSON, and XML.

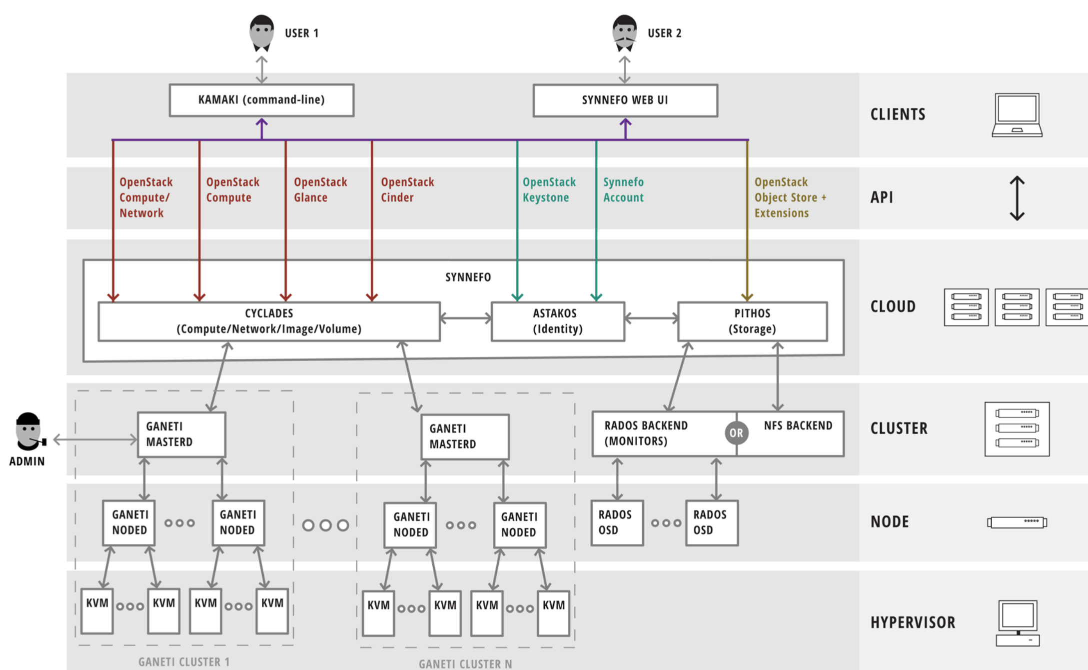


Figure 1: An overview of the Synnefo architecture including all layers

These two sides are served by people with different mindsets and different skill sets. We argue it also is most efficient to be served by different software, keeping a clear separation between the cloud and the cluster layers. Synnefo sits at the cloud layer. We wear a different hat when implementing Synnefo at the cloud layer than when implementing components at the cluster layer that integrate it with Ganeti, or when contributing to Ganeti itself.

Overall Architecture

An overview of the Synnefo stack is shown in Figure 1. Synnefo has three main components:

- Astakos is the common Identity/Account management service across Synnefo.
- Pithos is the File/Object Storage service.
- Cyclades is the Compute/Network/Image and Volume service.

Table 1 provides an explanation for the names we used.

These components are implemented in Python using the Django framework. Each service exposes the associated OpenStack APIs to end users. The service scales out on a number of workers, uses its own private DB to hold cloud-level data, and issues requests to the cluster layer, as necessary.

Synnefo has a number of smaller components that plug into Ganeti to integrate it into a Synnefo deployment.

In the following, we describe the functionality of each main component.

Astakos (Identity)

Astakos is the Identity management component, which provides a common user base to the rest of Synnefo. Astakos handles user creation, user groups, resource accounting, quotas, and projects, and it issues authentication tokens used across the infrastructure. Astakos supports multiple authentication methods: local username/password pairs; LDAP/Active Directory; SAML 2.0 (Shibboleth) federated logins; and login with third-party credentials, including Google, Twitter, and LinkedIn. Users can add

Synnefo	Greek for "cloud," which seemed good for a cloud platform.
~okeanos	Greek for "ocean," an abundant resource pool for life on Earth.
Astakos	Greek for "lobster," a crustacean with big claws and a hard exoskeleton.
Pithos	Ancient Greek name for storage vessels, e.g., for oil or grains.
Cyclades	The main island group in the Aegean Sea.
Kamaki	Greek for "harpoon"; if VMs are fish in the ocean, a harpoon may come handy.
Archipelago	Greek for "a cluster of islands," which seemed good for a distributed storage system.

Table 1: The story behind the names of Synnefo and its components. Many of the names follow a sea theme, as Synnefo's origins are in the ~okeanos service.

multiple login methods to a single account, according to configured policy.

Astakos keeps track of resource usage across Synnefo, enforces quotas, and implements a common user dashboard. Quota handling is resource-type agnostic: resources (e.g., VMs, public IPs, GBs of storage, or disk space) are defined by each Synnefo component independently, then imported into Astakos for accounting and presentation.

Astakos runs at the cloud layer and exposes the OpenStack Keystone API for authentication, along with the Synnefo Account API for quota, user group, and project management.

Pithos (Object/File Storage)

Pithos is the Object/File Storage component of Synnefo. Users upload files on Pithos using either the Web UI, the command-line client, or native syncing clients. Pithos is a thin layer mapping user-files to content-addressable blocks that are then stored on a storage backend. Files are split in blocks of fixed size, which are hashed independently to create a unique identifier for each block, so each file is represented by a sequence of block names (a *hashmap*). This way, Pithos provides deduplication of file data; blocks shared among files are only stored once. The current implementation uses 4 MB blocks hashed with SHA256. Content-based addressing also enables efficient two-way file syncing that can be used by all Pithos clients (e.g., the "kamaki" command-line client or the native Windows/Mac OS clients). Whenever someone wants to upload an updated version of a file, the client hashes all blocks of the file and then requests the server to create a new version for this block sequence. The server will return an error reply with a list of the missing blocks. The client may then upload each block one by one, and retry file creation. Similarly, whenever a file has been changed on the server, the client can ask for its list of blocks and only download the modified ones.

Pithos runs at the cloud layer and exposes the OpenStack Object Storage API to the outside world, with custom extensions for syncing. Any client speaking to OpenStack Swift can also be used to store objects in a Pithos deployment. The process of mapping user files to hashed objects is independent from the actual storage backend, which is selectable by the administrator using pluggable drivers. Currently, Pithos has drivers for two storage backends: files on a shared file system (e.g., NFS, Lustre, or GPFS) or objects on a Ceph/RADOS [3] cluster. Whatever the storage backend, it is responsible for storing objects reliably, without any connection to the cloud APIs or to the hashing operations.

Cyclades (Compute/Network/Image/Volume)

Cyclades is the Synnefo component that implements the Compute, Network, Image, and Volume services. Cyclades exposes the associated OpenStack REST APIs: OpenStack Compute, Network, Glance, and, soon, Cinder. Cyclades is the part that manages multiple Ganeti clusters at the backend. Cyclades issues commands to a Ganeti cluster using Ganeti's Remote API (RAPI). The administrator can expand the infrastructure dynamically by adding new Ganeti clusters to reach datacenter scale. Cyclades knows nothing about low-level VM management operations, e.g., handling of VM creations, migrations among physical nodes, and handling of node downtimes; the design and implementation of the end-user API is orthogonal to VM handling at the backend.

We strive to keep the implementation of Cyclades independent of Ganeti code. We write around Ganeti, and add no Synnefo-specific code inside it. Whenever the mechanism inside Ganeti does not suffice, we extend it independently from Synnefo, and contribute patches to the official upstream for review and eventual inclusion in the project.

There are two distinct, asynchronous paths in the interaction between Synnefo and Ganeti. The *effect* path is activated in response to a user request; Cyclades issues VM control commands to Ganeti over RAPI. The *update* path is triggered whenever the state of a VM changes, due to Synnefo- or administrator-initiated actions happening at the Ganeti level. In the update path, we exploit Ganeti's hook mechanism to produce notifications to the rest of the Synnefo infrastructure over a message queue.

Tying It All Together

Synnefo's greatest strength lies in the integrated way it handles its three basic storage entities: Files, named pieces of user data; Images, the static templates from which live VM instances are initialized; and Volumes, the block storage devices, the virtual disks on which live VMs operate. In this section, we describe the duality between Files and Images (an Image is a file on Pithos that has specific metadata), and the duality between Images and Volumes (a Volume is a live VM disk that originates from an Image).

Images as Files on Pithos

Synnefo uses Pithos to store both system and user-provided Images in the same way it stores all other files. Because Images of the same OS share many identical blocks, deduplication comes in handy. Assume a user has created a "golden" VM Image on her own computer, and has customized it to her liking. When she is ready to deploy it, she uploads it as a file to Pithos, registers it as an Image with Cyclades, then spawns new VMs from it. When she needs to update her Image, she just repeats the process. Every upload uses the Pithos syncing protocol, which means the client will only need to upload the blocks changed since the previous time. Pithos features a file-sharing mechanism, which applies to Image files too: users can attach custom ACLs to them, share them with other users or closed groups, or make them public.

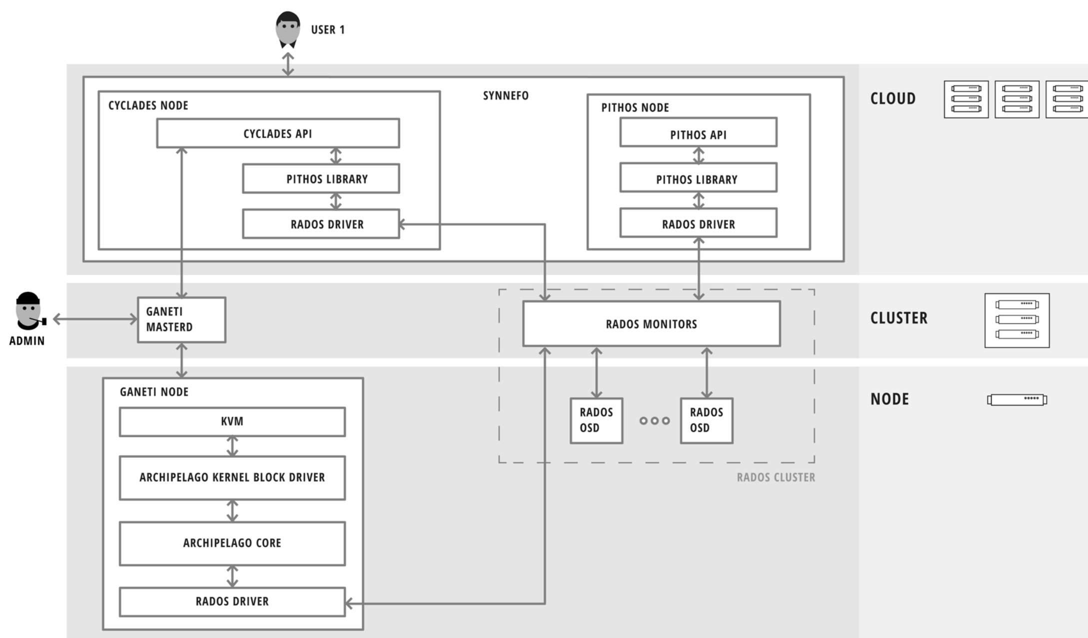


Figure 2: Integrated storage for Images and Volumes with Archipelago

Image Deployment Inside Ganeti

To support the secure deployment of user-provided, untrusted images with Ganeti, we have developed a Ganeti OS definition called *snf-image*. Image deployment entails two steps: (1) Volume initialization—the Image is fetched from backend storage and copied to the block device of the newly created instance, and (2) optional Image customization. Customization includes resizing the root file system, changing passwords for root or other users, file injection (e.g., for SSH keys), and setting a custom hostname. All Image customization is done inside a helper VM, in isolation from the physical host, enhancing robustness and security.

For Volume initialization, snf-image can fetch Image data from a number of storage backends. Volume initialization can use a shared file system (e.g., NFS), perform an HTTP or FTP download, or, in the Synnefo case, contact a Pithos storage backend directly. snf-image can deploy most major Linux distributions (Debian, Ubuntu, CentOS, Fedora, Arch, openSUSE, Gentoo), Windows Server 2008R2 and 2012, as well as FreeBSD.

Archipelago: Integrated Handling of Volumes and Images

Synnefo supports all different storage options ("disk templates") offered by Ganeti to back the virtual disks used by VMs ("Volumes"). Each storage backend has different redundancy and performance characteristics; Synnefo brings the choice of storage backend all the way up to the user, who can select based on the intended usage of the VM.

The Ganeti-provided disk templates are good options for long-running, persistent VMs (e.g., a departmental file server running on the cloud); however, they are not a good fit when the usage scenario needs thin VM provisioning: for example, when the user wants to spin up a large number of short-lived, identical VMs (e.g., from a custom golden Image), run a parallel program for a few hours or days, then shut them down. In this case, the time and space overhead of copying Image data to all Volumes is significant.

Archipelago is a block storage layer developed with Synnefo, which integrates VM Images with Volumes. Archipelago enables thin creation of Volumes as copy-on-write clones of Images, with zero data movement, as well as making snapshots of a Volume at a later time to create VM Images. Archipelago plugs into Ganeti and acts as one of its disk templates. Cyclades then uses Archipelago for fast provisioning of VMs from Images stored on Pithos, with minimal overhead. To implement clones and snapshots, Archipelago keeps track of VM block allocation in maps, initialized from Pithos files (hashmaps). Maps are stored along with actual data blocks. Archipelago can use various storage backends to store data, similarly to Pithos. Archipelago has pluggable drivers, currently for file system-backed block storage, or Ceph/RADOS, so clone and snapshot functionality is independent of the underlying backend. Figure 2 shows how Archipelago is integrated into a Synnefo deployment. In such a scenario, Archipelago shares its storage backend with Pithos. This enables a workflow as follows: a user uploads the contents of an Image as a file on Pithos, with efficient syncing, registers it as an Image

with Cyclades, then spawns a large number of thinly provisioned VMs from this Image. Because Archipelago shares the storage backend with Pithos, it creates one new volume per VM without copying the data. The actual 4 MB blocks of data that make up the Image remain as blocks in the storage backend, after being uploaded to Pithos by the user. Archipelago will create one map per VM, with all maps referencing the original Pithos blocks for the Image. Whenever a VM modifies data on its volume, Archipelago allocates a new block for it and updates the map for its volume accordingly.

Synnefo Advantages

The decoupled design of Synnefo brings the following advantages:

- Synnefo combines the stability of Ganeti with the self-service provisioning of clouds. This allows it to run workloads that do not fit the standard model of a volatile cloud, such as long-running servers in fault-tolerant, persistent VMs. Archipelago-backed storage covers the need for fast provisioning of short-lived, computationally intensive worker VMs.
- In a Synnefo deployment, Synnefo and Ganeti follow distinct upgrade schedules, with software upgrades rolled out gradually, without affecting all of the stack at once.
- The Ganeti clusters are self-contained. The administrator has complete control (e.g., to add/remove physical nodes or migrate VMs to different nodes via the Ganeti side path) without Synnefo knowing about it. Synnefo is automatically notified and updates user-visible state whenever necessary. For example, a VM migration happening at Ganeti level is transparent to Synnefo, whereas a VM shutdown by the admin will propagate up to the user.
- The system scales dynamically and linearly by adding new Ganeti clusters into an existing installation. Heterogeneity across clusters allows Synnefo to provide services with different characteristics and levels of QoS (e.g., virtual-to-physical CPU ratio).
- Two-level allocation policy for VMs with different criteria: at the cloud layer, Synnefo selects a Ganeti cluster according to high-level criteria (e.g., QoS); at the cluster layer, Ganeti selects a physical node based on lower-level criteria (e.g., free RAM on node).
- There is no single database housing all VM configuration data. Low-level state is handled separately in each Ganeti cluster. Physical nodes have no access to the Cyclades database at the cloud layer. This minimizes the possible impact of a hypervisor breakout and simplifies hardening of DB security.
- Out–of-the-box integration with different storage backend technologies, including File, LVM, DRBD, NAS, or Archipelago on commodity hardware.

Running in Production

Synnefo has been running in production since 2011, powering GRNET's ~okeanos [1] public cloud service. Synnefo's development team has grown to more than 15 people in the past three years. As of this writing, ~okeanos runs more than 5,000 active VMs, for more than 3,500 users. Users have launched more than 100,000 VMs and more than 20,000 virtual networks.

Using Synnefo in production has enabled:

- Rolling software and hardware upgrades across all nodes. We have done numerous hardware and software upgrades (kernel, Ganeti, Synnefo), many requiring physical node reboots, without user-visible VM interruption.
- Moving the whole service to a different datacenter, with cross-datacenter live VM migrations, from Intel to AMD machines, without the users noticing.
- On-the-fly syncing of NFS-backed Pithos blocks to RADOS-backed storage, and integration with Archipelago for thin VM provisioning.
- Scaling from a few physical hosts to multiple racks with dynamic addition of Ganeti backends.
- Overcoming limitations of the networking hardware regarding number of VLANs. Ganeti provides for pluggable networking scripts, which we exploit to run thousands of virtual LANs over a single physical VLAN with MAC-level filtering, in a custom configuration. We have also tested VXLAN-based network encapsulation, again with no code modifications to Ganeti or Synnefo.
- Preserving the ability to live migrate while upgrading across incompatible KVM versions, by maintaining the virtual hardware configuration independently.

Synnefo is open source. Source code, distribution packages, documentation, many screenshots and videos, as well as a test deployment open to all can be found at http://www.synnefo.org.

References

[1] Vangelis Koukis, Constantinos Venetsanopoulos, and Nectarios Koziris, "~okeanos: Building a Cloud, Cluster by Cluster," *IEEE Internet Computing*, vol. 17, no. 13, May-June 2013, pp. 67-71.

[2] Guido Trotter and Tom Limoncelli, "Ganeti: Cluster Virtualization Manager," USENIX *;login:*, vol. 38, no. 3, 2013.

[3] Sage A. Weil, Andrew W. Leung, Scott A. Brandt, and Carlos Maltzahn, "RADOS: A Scalable, Reliable Storage Service for Petabyte-Scale Storage Clusters," *Proceedings of the 2nd International Workshop on Petascale Data Storage*, PDSW '07, held in conjunction with Supercomputing '07 (ACM, 2007), pp. 35-44.

vPipe: One Pipe to Connect Them All

SAHAN GAMAGE, RAMANA KOMPELLA, AND DONGYAN XU

Sahan Gamage is a Ph.D. candidate in the Computer Science Department at Purdue University and is advised by Professors Dongyan Xu and Ramana Kompella. His research focuses on improving I/O performance in virtual machines in cloud environments. Sahan received an M.S. in Computer Science from Purdue University and a B.S. in Computer Science from University of Moratuwa, Sri Lanka. sgamage@purdue.edu

Ramana Kompella is an Associate Professor in the Computer Science Department at Purdue University. He directs the Systems and Networking (SYN) Lab at Purdue, conducting research on various networking research problems in cloud computing, virtualization, datacenter networking, and software-defined networking. Before coming to Purdue, he obtained his Ph.D. from UCSD. rkompella@purdue.edu

Dongyan Xu is a Professor and University Faculty Scholar in the Computer Science Department at Purdue University. He leads the FRIENDS Lab at Purdue, conducting research in virtualization technologies, cloud computing, and computer systems security and forensics. He received a Ph.D. in Computer Science from the University of Illinois at Urbana-Champaign in 2001. dxu@cs.purdue.edu

Many enterprises use the cloud to host applications such as Web services, big data analytics, and storage, which involve significant I/O activities, moving data from a source to a sink, often without even any intermediate processing; however, cloud environments tend to be virtualized, which introduces a significant overhead for I/O activity as data needs to be moved across several protection boundaries. CPU sharing among virtual machines (VMs) introduces further delays into the overall I/O processing data flow. In this article, we present an abstraction called vPipe to mitigate these problems. vPipe introduces a simple "pipe" that can connect data sources and sinks, which can be either files or TCP sockets, at the virtual machine monitor (VMM) layer. Shortcutting the I/O at the VMM layer achieves significant CPU savings and avoids scheduling latencies that degrade I/O throughput.

Cloud computing platforms such as Amazon EC2 support a large number of real businesses hosting a wide variety of applications. For instance, several popular companies (e.g., Pinterest, Yelp, Netflix) host large-scale Web services on the EC2 cloud. Many enterprises (e.g., Foursquare) also use the cloud for running analytics and big data applications using the MapReduce framework. Companies such as Dropbox also use the cloud for storing customers' files. While these applications are quite diverse in their functionality and the services they offer, they share one common characteristic: they all involve a significant number of I/O activities, moving data from one I/O device (source) to another (sink). The source or sink can be either the network or the disk and typically varies across applications (see Table 1). Although an application may sometimes process or modify data after it reads from the source and before it writes to the sink, in many cases it may merely relay the data without any processing.

Meanwhile, cloud environments use virtualization to achieve high resource utilization and strong tenant isolation. Thus, cloud applications/services are executed in virtual machines that are multiplexed over multiple cores of physical machines. Further, there is a lot of variety in the CPU resources offered to individual VMs. For instance, Amazon EC2 supports small, medium, large, and extra large instances, which are assigned 1, 2, 4, and 8 EC2 compute units, respectively, with each EC2 unit roughly equivalent to a 1 GHz core [1]. Because modern commodity cores run at 2–3 GHz, a core may be shared by more than one instance.

Now, imagine running the above I/O-intensive applications in such CPU-sharing instances in the cloud. As an example, let us focus on a simple Web application that receives an HTTP request from a client that results in reading a file from the disk and then writing it to a network socket. The flow of data, as shown in Figure 1(a), involves reading the file's data blocks into the application after they cross the VMM and the guest kernel boundaries, and then writing them into the TCP socket, causing the data to pass again through the same protection boundaries before reaching the physical NIC.

There are two main problems with this simple data flow model. First, transferring data across all the protection layers incurs significant CPU overhead, which affects the cloud provider (provisions more CPU for hypervisor) as well as the tenant (costs more for the job).

Application	Data Source	Data Sink
Web server hosting static files	Disk	TCP socket
User uploading a file to cloud storage	TCP socket	Disk
File backup service	Disk	Disk
Web proxy server or a load balancer	TCP socket	TCP socket

Table 1: I/O sources and sinks for typical cloud applications

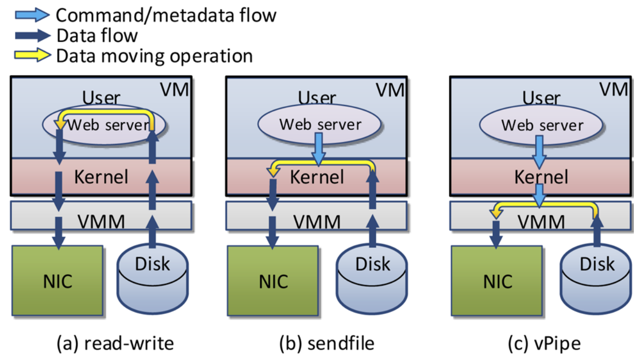


Figure 1: I/O data flow for a Web server

Using zero-copy system calls such as mmap() and sendfile() in the guest VM, as shown in Figure 1(b), would clearly reduce the copy overhead to some extent, but not by much, because the major portion of the overhead (e.g., virtual interrupts, protection domain switching) is actually incurred when data crosses the VMM-VM boundary. Second, and perhaps more importantly, because of CPU sharing with other VMs, this VM may not always be scheduled, which will introduce delays in the data flow, resulting in significant degradation of performance. For more information about how VM scheduling affects TCP, refer to [5] and [3].

With vPipe, we propose a new abstraction to address both of these problems—i.e., eliminate CPU overhead and reduce I/O processing delay—in virtualized clouds with CPU sharing. The key idea of vPipe is to empower the VMM to "pipe" data directly from the source to the sink without involving the guest VM. As shown in Figure 1(c), vPipe incurs fewer copies across protection boundaries, and completely eliminates the more costly VMM-VM data transfer overhead, thereby reducing CPU usage significantly, which in turn saves money for both the cloud provider and the tenant. Furthermore, because the VMM is often running in a dedicated core, any scheduling latencies experienced by the guest VM due to CPU sharing have virtually no impact on I/O performance.

Although our idea of vPipe makes intuitive sense, realizing it is not that straightforward because the meta-information regarding the source and sink of a "vPipe" resides in the VM context. We need to create a new interface to enable the application to, with support of the guest kernel, pass this information to the VMM and instruct it to create the source-sink pipe. For example, the VM needs to identify the physical block identifiers of the file and establish the TCP socket, which can then be passed down to the VMM layer for establishing the pipe. For applications that insert new data into the data stream, there also must be sufficient flexibility in vPipe to allow the VMM and VM to take control of the pipe. For example, HTTP responses are typically preceded by an HTTP response header; so the Web server first needs to write the HTTP header to the sink (i.e., TCP socket), call vPipe to transfer control to the VMM layer to pipe the file to the TCP socket, and then transfer the control back (e.g., to keep the connection alive for persistent HTTP).

We describe how vPipe works and show the effectiveness of vPipe using a proof-of-concept implementation of a simple disk-to-network vPipe in Xen/Linux with the example of a Web server serving static files to clients.

Creating an I/O Shortcut at VMM

The key idea behind vPipe is to create an I/O data "shortcut" at the VMM layer when an application needs to move data from one I/O device to another. We essentially expose a set of new library calls (e.g., vpipe_file() similar to the UNIX sendfile()) to enable applications to create and manage this I/O shortcut. Implementing these new calls (shown in Figure 2) requires support at the guest kernel and the VMM layer, which are provided by two main components: (1) vPipe-vm for support in the guest kernel; and (2) vPipe-drv for support in the driver domain (VMM layer). Coordination across the driver domain-VM boundary is achieved with the help of a standard inter-domain channel (e.g., Xen uses ring buffers and event channels) that exist in any virtualized host.

Initially, when we activate vPipe from inside the VM, the vPipe-vm module registers a special device in the system, /dev/vpdev, that facilitates communication between the user process and the guest kernel via ioctl() function. This step is designed to prevent introducing a new system call, which would in turn require modifications to the guest kernel.

There are four main steps involved in vPipe-enabled I/O. First, the application running inside the VM invokes the corresponding vPipe call with source and sink file/socket descriptors and blocks (we can also implement a non-blocking version of this) until it is completed. Second, the vPipe-vm component validates the file/socket descriptors and dereferences them to obtain the corresponding information about them (e.g., block IDs, socket structures) that is then passed on to the driver domain. Third, the vPipe-drv component uses this information and performs

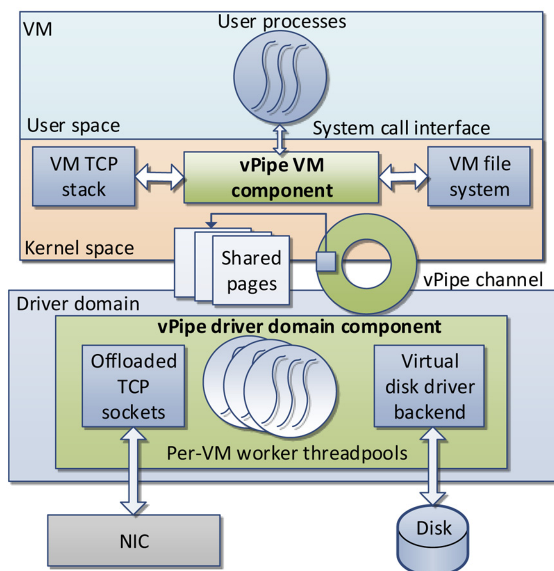


Figure 2: vPipe architecture

the actual "piping" operation. Finally, upon completion, the driver domain component notifies the guest OS with information about the data transfer through the inter-domain channel. The guest OS then passes the notification back to the application unblocking the call.

Offloading a TCP Connection

If the vPipe source/sink is an established TCP connection, we offload the entire TCP connection to the driver domain by supplying essential socket details (such as IP addresses, ports, and sequence numbers) and letting the TCP stack at the driver domain perform TCP processing as long as vPipe exists. Most VMMs have a fully functional TCP stack to carry out management tasks, and we use this for our offloaded TCP sockets.

When a TCP socket is used as either end of a vPipe, we first use the guest OS virtual file system (VFS) to translate the file descriptor to the kernel socket structure and collect the socket information (TCP 4-tuple, sequence numbers, and congestion control information). We reuse congestion control information from the VM's socket to initialize the vPipe socket at the driver domain, instead of restarting it from slow start. This information is then passed on to vPipe-drv.

Upon receiving this information, vPipe-drv creates a TCP socket using the driver domain's TCP stack; however, we do not use system calls such as connect() on this socket; we instead instantiate the kernel socket structure of the new socket using the original connection's metadata from vPipe-vm. There is an additional issue we need to address: the need to add a static route entry in the driver domain's IP routing table to route the packets to the local TCP/IP stack if the packets match the 4-tuple described above, otherwise they will go directly to the guest VM.

Finally, we mark the socket as "established," which informs the driver domain's TCP stack that the socket is ready to receive packets. vPipe-drv can then perform standard socket operations such as send() and recv() on this socket.

Offloading a File I/O Operation

If the vPipe source/sink is a file, similar to the socket, we use VM's file system to obtain metadata about the file data blocks and transfer this information to the driver domain where either the reading or writing of the data blocks is carried out. Unlike TCP packets, file metadata is stored separately from the actual data, in the form of separate disk blocks (e.g., inode blocks). Once the metadata is passed on from the VM level, for the driver domain to access the corresponding file by simply using the physical block identifiers is straightforward.

When the source of a vPipe is a file, vPipe-vm will first locate the file's inode using the file descriptor. Then vPipe-vm uses file system-specific functions and device information from the inode to obtain the file's physical data block identifiers. This information is then encapsulated in a vPipe custom data structure, along with number of bytes to read and offset of the first byte to transfer, and passed to the driver domain via the communication channel.

Once vPipe-drv receives this information, it prepares a set of block I/O operation descriptors using a preallocated set of pages and the block identifiers supplied by vPipe-vm and submits them to the emulated block device.

Writing to a file involves either creating a new file, appending to an existing file, or overwriting an existing file. When overwriting a file, we can use the same method as reading the file to get the file block identifiers. But when we are creating a new file or appending to an existing file, we need to request the guest's file system to create new block identifiers for new data. This is done by vPipe-vm requesting the guest file system to create or update the inode for the new data blocks with an empty set of data blocks. This will generate a new set of block identifiers that will be transferred to vPipe-drv, where the actual writing of the data blocks will be performed.

Connecting the Dots

When vPipe-drv receives a vPipe request from the VM, it creates a "pipe descriptor" associated with that operation. This descriptor contains metadata describing each source/sink and two functions: a read function that implements one of the above read strategies, and a write function that implements one of the write strategies depending on the source and the sink. A free thread is picked up from the thread pool, and this thread will call the read function using the source's metadata. As data returns from the source, the thread will call the write function to output the data using the metadata of the sink.

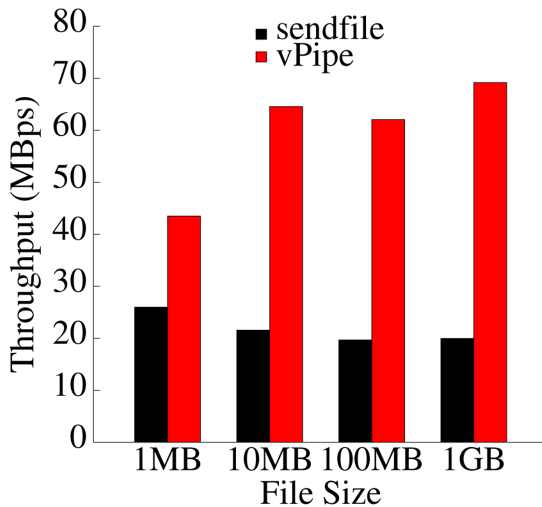


Figure 3: lighttpd throughput improvement

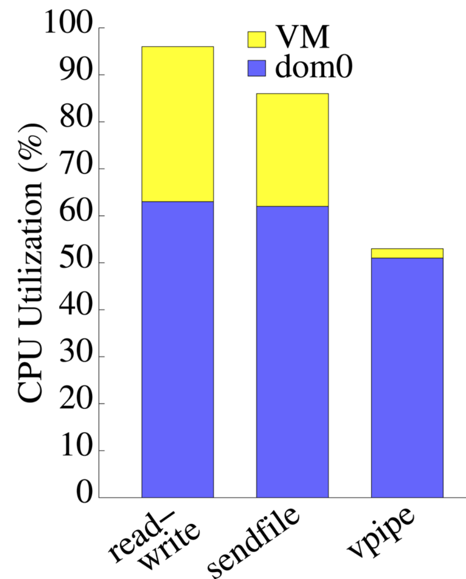


Figure 4: CPU savings by vPipe

Sharing the Driver Domain

vPipe poses one more challenge: because the actual I/O operations are performed by vPipe-drv, we should "charge" the work done by the worker threads in the driver domain (Figure 2) to the VMs requesting vPipe-enabled I/O. Lack of driver domain access accounting and control will lead to unfairness among the requesting VMs. To address this problem, we propose a simple credit-based system. Each VM-specific thread pool in the driver domain is allocated a certain amount of credits based on the priority (weight) of the VM. As the threads execute, they consume the allocated credits based on the number of bytes transferred. When the credits run out, the corresponding worker threads will block until a timer task adds more credits to them.

vPipe on Xen/Linux

We implemented a prototype of vPipe on Xen 4.1 as the virtualization platform and Linux 3.2 as the kernel of VMs and the driver domain. vPipe-vm is implemented as a loadable kernel module. Because it uses standard Linux VFS functions already exposed to kernel modules to manipulate file descriptors and sockets, vPipe-vm requires no changes to the guest kernel. This makes vPipe-vm attractive for customers, because no kernel recompilation is required for using vPipe.

We add a similar loadable kernel module in Xen's driver domain to implement vPipe-drv; however, we must make a few small changes in the main kernel code, such as adding special functions to create offloaded sockets and adding static routes.

We implement the driver domain-VM communication channel as a standard Xen device with a ring buffer and an event channel.

Improved lighttpd Throughput

lighttpd [2] is a highly scalable lightweight Web server that we adapt to vPipe. To do so, we just replaced "sendfile()" with "vpipe_file()" in the lighttpd source code and recompiled it. Figure 3 shows the average I/O throughput reported by httperf for different file sizes, when the VM running lighttpd is co-located with two other VMs. Whereas lighttpd using vPipe shows throughput improvement for all file sizes tested, improvement for larger files tends to be greater (up to 3.4×). For smaller files, the overhead of offloading the connection and the file block information to the driver domain affects the overall time, and hence the throughput improvement is comparatively less than that for large files.

CPU Savings by vPipe

Figure 4 shows the overall average CPU utilization of both the driver domain and the VM when transferring a 1 GB file. As expected, the VM's CPU utilization for read-write mode is the highest because it requires copying data across all layers. The sendfile() system call eliminates the kernel to userland copying and, hence, its VM CPU utilization is less than that of the read-write mode. vPipe incurs the least CPU utilization at VM level because there is no work to be done in the VM context once the operation is offloaded to the driver domain.

With vPipe offloading the I/O processing task to the driver domain, we would expect that the driver domain CPU utilization for vPipe mode would be the highest. (Somewhat) surprisingly, this is not the case, as shown in Figure 4. This is because, with vPipe, we eliminate the data processing by the device emulation

layer at the driver domain, which is required to transfer disk blocks and network packets to and from the VM in the other two modes.

Wrapping Up

vPipe is a new I/O interface for applications in virtualized clouds, which mitigates virtualization-related performance penalties by shortcutting I/O operations at the VMM layer. Our experiments with the vPipe prototype shows that vPipe can improve lighttpd I/O throughput while reducing CPU utilization. vPipe also requires minimal modifications to existing applications, such as Web servers, and facilitates a simple deployment. You can find more information about vPipe in [4].

References

[1] Amazon EC2 instance types: http://aws.amazon.com/ec2/instance-types/.

[2] lighttpd Web server: http://www.lighttpd.net/.

[3] S. Gamage, A. Kangarlou, R. R. Kompella, and D. Xu, "Opportunistic Flooding to Improve TCP Transmit Performance in Virtualized Clouds," ACM SOCC (2011).

[4] S. Gamage, R. R. Kompella, and D. Xu, "vPipe: One Pipe to Connect Them All!" USENIX HotCloud (2013): https://www.usenix.org/conference/hotcloud13/vpipe-one-pipe-connect-them-all.

[5] A. Kangarlou, S. Gamage, R. R. Kompella, and D. Xu, "vSnoop: Improving TCP Throughput in Virtualized Environments via Acknowledgement Offload," ACM/IEEE SC (2010).

USENIX Member Benefits

Members of the USENIX Association receive the following benefits:

Free subscription to *;login:*, the Association's bi-monthly print magazine, and *;login: logout*, our Web-exclusive bi-monthly magazine. Issues feature 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 the current 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/member-services/discounts

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-services or contact office@usenix.org. Phone: 510-528-8649

Hyper-Switch: A Scalable Software Virtual Switching Architecture

KAUSHIK KUMAR RAM, ALAN L. COX, AND SCOTT RIXNER

Kaushik Kumar Ram recently graduated from Rice University with a Ph.D. in Computer Science. In his graduate research work, he explored new mechanisms and architectures for the network subsystem in virtualized systems. He likes to build systems software to solve interesting problems in the areas of operating systems and networking. He received his B.Tech in Computer Science and Engineering from Indian Institute of Technology in Guwahati, India. kaukum@gmail.com

Alan L. Cox is an Associate Professor of Computer Science at Rice University and a long-time contributor to the FreeBSD project. Over the years, his research has sought to address fundamental problems at the intersection of operating systems, computer architecture, and networking. Prior to joining Rice, he earned his B.S. at Carnegie Mellon University and his Ph.D. at the University of Rochester. alc@rice.edu

Scott Rixner is an Associate Professor of Computer Science at Rice University. His research focuses on the interaction between operating systems, runtime systems, and computer architectures; memory controller architectures; and hardware and software architectures for networking. He works with both large server-class systems and small embedded systems. Prior to joining Rice, he received his Ph.D. from MIT. rixner@rice.edu

In virtualized datacenters, the last hop switching happens inside a server. In this article we describe the Hyper-Switch, a highly efficient and scalable software-based network switch that works alongside driver domains. Hyper-Switch outperforms existing virtual switches used in Xen and KVM, especially for inter-VM network traffic, and this performance will soon be critical in datacenters.

Machine Virtualization in Datacenters

Machine virtualization has become a cornerstone of modern datacenters; it enables server consolidation as a means to reduce costs and increase efficiencies. Many cloud-based service infrastructures use machine virtualization as one of their fundamental building blocks. Further, it is also being used to support the utility computing model where users can "rent" time in a large-scale datacenter. These benefits of machine virtualization are now widely recognized. Consequently, the number of virtual servers in production is rapidly increasing.

The use of machine virtualization has led to considerable change to the datacenter network. In particular, the communication endpoints within the datacenter are now virtual machines (VMs), not physical servers. Consequently, the datacenter network now extends into the server, and last hop switching occurs inside the physical server. In other words, a virtual switch within the server is ultimately responsible for demultiplexing and forwarding packets to their destinations.

Communication between servers within the same datacenter already accounts for a significant fraction of a datacenter's total network traffic [3]. Moreover, a recent study of multiple datacenter networks reported that 80% of the traffic originating at servers in cloud datacenters never leaves a rack [1]. Further, the number of cores on a chip is predicted to grow to 64 in a few years and to 256–512 by the end of the decade [2]. If this prediction comes to pass, then a rack of servers may be replaced by VMs in a single physical server, and the network traffic that today never leaves a rack may instead never leave a server. These datacenter trends necessitate the need for a high-performance virtual switch to support efficient communication—especially between VMs—in virtualized servers.

Software Virtual Switching Solutions

There are many I/O architectures for network communication in virtualized systems. Of these, software device virtualization is most widely used. This preference for software over specialized hardware devices is due in part to the rich set of features—including security, isolation, and mobility—that the software solutions offer. The software solutions can be further divided into driver domain and hypervisor-based architectures. Driver domains are dedicated VMs that host the drivers used to access the physical devices; they provide a safe execution environment for the device drivers.

Arguably, hypervisors that support driver domains are more robust and fault tolerant, as compared to the alternate solutions that locate the device drivers within the hypervisor. This is becoming an important requirement, especially as servers in datacenters move toward multi-tenancy; however, this reliability comes at a price because the use of driver domains

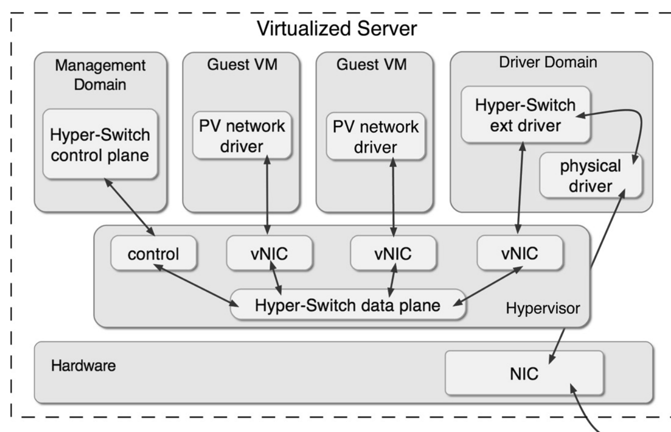


Figure 1: The Hyper-Switch architecture

leads to significant software overheads that not only reduce the achievable I/O performance but also severely limit I/O scalability [8]. Specifically, the sharing of I/O buffers between the driver domain and guest VMs is expensive because it requires hypervisor intervention to maintain memory isolation.

There are fundamental problems with traditional driver domain architectures. Essentially, the driver domain must be scheduled to run whenever packets are waiting to be processed. As a result, scheduling overheads are incurred while processing network packets. Further, the driver domain must be scheduled in a timely manner to avoid unpredictable delays in the processing of network packets, which is very hard to achieve for all workloads.

In real-world virtualization deployments , dedicating processor cores to the driver domain is standard practice . This avoids scheduling delays but often leaves cores idle. In fact, dedicating CPU resources for backend processing is not limited to just driver domain-based architectures (e.g., SplitX [4]); however, this can lead to underutilization of these cores. This goes against one of the fundamental tenets of virtualization: to enable the most efficient utilization of the server resources.

Hyper-Switch

We explored the virtual switching design space to see whether we could achieve both high-performance and fault tolerance at the same time. If you look at existing I/O architectures, the virtual switch is implemented inside the same software domain where the virtual devices are implemented and the device drivers are hosted. For instance, all these components are implemented inside a driver domain in Xen and the host OS in KVM. This colocation is purely a matter of convenience because packets must be switched when they are moved between the virtual devices and the device drivers.

We introduce the Hyper-Switch [7], which challenges the existing convention by separating the virtual switch from the domain that hosts the device drivers. The Hyper-Switch is a highly efficient and scalable software switch for virtualization platforms that support driver domains. In particular, the hypervisor includes the data plane of a flow-based software switch, while the driver domain continues to safely host the device drivers.

Figure 1 illustrates the Hyper-Switch architecture. In Hyper-Switch, the hypervisor implements just the data plane of the virtual switch that is used to forward network packets between VMs. The switch's control plane is implemented in the management layer. Incoming external network traffic is initially handled by the driver domain because it hosts the device drivers, and then is forwarded to the destination VM through Hyper-Switch. For outgoing external traffic, these two steps are reversed. So the virtual switch implementation is distributed across virtualization software layers with only the bare essentials implemented inside the hypervisor. The separation of control and data planes is achieved using a flow-based switching approach. This is similar to how switching is performed using OpenFlow [5].

Basic Design

Packet processing by Hyper-Switch begins at the transmitting VM (or driver domain) where the packet originates and ends at the receiving VM (or driver domain) where the packet has to be delivered. Packet processing proceeds in four stages:

1. **Packet transmission.** In the first stage, the transmitting VM pushes the packet to the Hyper-Switch for processing. Packet transmission begins when the guest VM's network stack forwards the packet to its paravirtualized network driver. Then the packet is queued for transmission by setting up descriptors in the transmit ring.
2. **Packet switching.** In the second stage, the packet is switched to determine its destination. Switching is triggered by a hypercall from the transmitting VM and begins with reading the transmit ring to find new packets. Each packet is then pushed to Hyper-Switch's data plane where it is switched using the flow-based approach. The data plane must be able to read the packet's headers in order to switch it. Because the data plane is located in the hypervisor, which has direct access to every VM's memory, it can read the headers directly from the transmitting VM's memory.
3. **Packet copying.** In the third stage, the switched packet is copied into the receiving VM's memory. By default, the destination VM is responsible for performing packet copies. Once switching is completed, the destination VM is notified via a virtual interrupt. Subsequently, that VM issues a hypercall. While in the hypervisor, the VM copies the packet into its memory. Note that the packet is copied directly from the transmitting VM's memory to the receiving VM's memory.

4. **Packet reception.** In the fourth and final stage, the paravirtualized network driver in the destination VM pushes the newly received packet into its network stack, where the packet is processed and eventually handed to some application. Note that the destination VM is already notified in the previous stage. So packet reception can happen as soon as the hypercall for copying the packet is complete.

Optimizations

Another important contribution of this work is a set of optimizations that increase performance. They enable Hyper-Switch to support both bulk and latency sensitive network traffic efficiently. They include:

- **Preemptive packet copying.** Packet copies are performed by default in a receiving VM's context; however, delivering a notification to a VM already requires entry into the hypervisor. So packet copy is performed preemptively when the receiving VM is being notified. In essence, the packet copy operation is combined with the notification to the receiving VM. This optimization avoids one hypervisor entry for every packet that is delivered to a VM.
- **Batching hypervisor entries.** In the Hyper-Switch architecture, as described thus far, the transmitting VM enters the hypervisor every time there is a packet to send. Moreover, the receiving VM is notified every time there is a packet pending in the internal receive queue. To mitigate these overheads, we use VM state-aware batching, which amortizes the cost of entering the hypervisor across several packets. This approach to batching shares some features with the interrupt coalescing mechanisms of modern network devices. Typically, in network devices, the interrupts are coalesced irrespective of whether the host processor is busy or not. But, unlike those devices, Hyper-Switch is integrated within the hypervisor, where it can easily access the scheduler to determine when and where a VM is running. So a blocked VM can be notified immediately when there are packets pending to be received by that VM. This enables the VM to wake up and process the new packets without delay. On the other hand, the notification to a running VM may be delayed if it was recently interrupted.
- **Offloading packet processing.** In Hyper-Switch, by default, packet switching is performed in the transmitting VM's context and packet copying is performed in the receiving VM's context. As a result, asynchronous packet switching does not occur with respect to the transmitting VM, and asynchronous packet copying does not occur with respect to the receiving VM; however, concurrent and asynchronous packet processing can significantly improve performance.

  Concurrent packet processing can be achieved by polling all the internal receive queues for packets waiting to be copied and polling all the transmit rings for packets waiting to be switched. This can be performed by processor cores that are currently idle. In this scheme, packet copying is prioritized over switching because packet copying is typically the more expensive operation, and a receiving VM is more likely to be performance bottlenecked than a transmitting VM.

  The idle cores are woken up just when there is work to be done. On the receive side, this can be ascertained precisely when switched packets are pending to be copied at a VM. Then one of the idle cores is chosen and woken up to perform the packet copies. A low-overhead mechanism is used to offload work to the idle cores. Note that this mechanism neither involves the scheduler nor requires any context-switching; instead, it uses a simple interprocessor messaging facility to directly request a specific idle core to copy packets to the VMs. Also, this mechanism attempts to spread the work across many idle cores to increase concurrency. Further, the offload mechanism is tuned to take advantage of CPU cache locality.

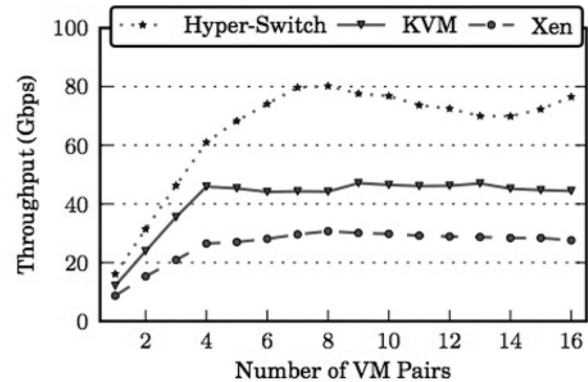


Figure 2: Pairwise performance scalability results

These optimizations enable efficient packet processing, better utilization of the available CPU resources, and higher concurrency. In particular, they take advantage of Hyper-Switch data plane's integration within the hypervisor and its proximity to the scheduler. As a result, Hyper-Switch enables much improved and scalable network performance, while maintaining the robustness and fault tolerance that derive from the use of driver domains.

Evaluation

We built a prototype of the Hyper-Switch architecture in the Xen virtualization platform. Here the switch's data plane was implemented by porting parts of Open vSwitch [6] to the Xen hypervisor. Open vSwitch's control plane was used without modification. We also developed a new paravirtualized network interface for the guest VMs to communicate with the data plane. The same interface was also used by the driver domain to forward external network traffic.

Then we evaluated Hyper-Switch using this prototype in Xen. The primary goal of this evaluation was to compare Hyper-Switch with existing architectures that implement the virtual switch either entirely within the driver domain or entirely within the hypervisor. To achieve this, the end-to-end performance under Hyper-Switch was compared to that under Xen's default driver domain-based architecture and KVM's hypervisor-based architecture. The evaluation showed that Hyper-Switch's performance was superior in terms of absolute bandwidth as well as scalability as the number of VMs and traffic flows were varied. Figure 2 shows the results from the pairwise scalability experiments, where the number of VM pairs was scaled up. Here, on a 32-core AMD machine, Hyper-Switch achieved a peak net throughput of ~ 81 Gbps as compared to only ~ 31 Gbps and ~ 47 Gbps under Xen and KVM, respectively. Interested readers are referred to our USENIX publication that includes more results from the evaluation [7].

Conclusion

In this work, we designed Hyper-Switch, which combines the best of the existing last hop virtual switching architectures. It hosted the device drivers in a driver domain to isolate any faults and the last hop virtual switch in the hypervisor to perform efficient packet switching. In particular, the hypervisor implemented just the fast, efficient data plane of a flow-based software switch. The driver domain was needed only for handling external network traffic.

We also implemented several carefully designed optimizations that enabled efficient packet processing, better utilization of the available CPU resources, and higher concurrency. As a result, the Hyper-Switch enabled much improved and scalable network performance, while maintaining the robustness and fault tolerance that derives from the use of driver domains. We believe that these optimizations should be a part of any virtual switching solution that aims to deliver high performance. The Hyper-Switch architecture demonstrates that it is feasible to switch packets between VMs at high-speeds without sacrificing reliability.

References

[1] T. Benson, S. Akella, and D. A. Maltz, "Network Traffic Characteristics of Data Centers in the Wild," IMC (2010).

[2] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger, "Dark Silicon and the End of Multicore Scaling," *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11 (ACM, 2011), pp. 365-376.

[3] A. Greenberg, J. Hamilton, D. A. Maltz, and P. Patel, "The Cost of a Cloud: Research Problems in Data Center Networks," *SIGCOMM Computer Communication Review*, vol. 39, no. 1 (2009), pp. 68-73.

[4] A. Landau, M. Ben-Yehuda, and A. Gordon, "SplitX: Split Guest/Hypervisor Execution on Multi-Core," WIOV '11: Proceedings of the 4th Workshop on I/O Virtualization (May 2011).

[5] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: Enabling Innovation in Campus Networks," *SIGCOMM Computer Communication Review*, vol. 38, no. 2 (April 2008), pp. 69-74.

[6] B. Pfaff, J. Pettit, T. Koponen, K. Amidon, M. Casado, and S. Shenker, "Extending Networking into the Virtualization Layer," HotNets-VIII: Proceedings of the Workshop on Hot Topics in Networks (October 2009).

[7] K. K. Ram, A. Cox, M. Chadha, and S. Rixner, "Hyper-Switch: A Scalable Software Virtual Switching Architecture," ATC '13: Proceedings of the USENIX Annual Technical Conference (June 2013).

[8] K. K. Ram, J. R. Santos, Y. Turner, A. L. Cox, and S. Rixner, "Achieving 10 Gb/s Using Safe and Transparent Network Interface Virtualization," *VEE '09: Proceedings of the ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (March 2009), pp. 61-70.

SYSADMIN

Technical Leadership Is Something We Can All Do

TOM LIMONCELLI

Tom is an internationally recognized author, speaker, and system administrator. His best known books include *Time Management for System Administrators* (O'Reilly) and *The Practice of System and Network Administration* (Addison-Wesley). In 2005 he received the SAGE Outstanding Achievement Award. He is an SRE at StackExchange.com. www.EverythingSysadmin.com is his blog. tal@everythingsysadmin.com

You don't have to be a team lead or manager to demonstrate leadership. Everyone on a sysadmin team can and should be a technical leader. To me, the children's game "Follow the Leader" exhibits the two essential qualities that are required to be a technical leader.

"Follow the leader" is a game that young children play. One child is selected to be "the leader" and walks as everyone else follows the leader in a single file line.

The leader might walk around the yard or playground, under a swing, between two big rocks. Everyone follows. If the leader takes a big leap over a big rock, everyone else leaps over it the same way. If the leader hops on one foot across a patio, everyone else hops on one foot across the patio. If there is a low-hanging branch obstructing the path, the leader lifts the branch up and walks under, then hands it to the next person, who hands it to the person behind them, and so on.

The leader is doing two essential things. These two things are the most essential parts of being a leader.

1. Go first.
2. Make it easy for others to follow.

They go first. There they are at the front of the line. The game doesn't work if they aren't.

They make it easy for others to follow. When coming to the low-hanging branch they might crawl under it, brush up against it, or lift it out of the way. The leader decides to lift it out of the way. By demonstrating how it is done, the leader makes it easy for others to follow. By making it easy, others can and do follow.

This is a powerful lesson about leadership. If you want to lead, you can't just "encourage" others to do things, you have to do that thing. You have to show that it is possible through action. If you want to lead, you have to make it easy for others to follow. You have to provide the training, the knowledge. You have to clear the path.

If you only do one of those things and not the other, things fall apart. If you aren't willing to go first, people will not follow. If you don't make it easy for others to follow, they'll do something else that is easier instead.

When I was a little boy I was told it was polite to let guests go first. That was true for serving food, but nobody told me it wasn't true for everything else. Trying to lead without going first is a disaster. When leading a group of guests through the house, I'd let the guests go through a door first. Now they're in front and don't know where to go and I'm trying to catch up. It is chaos. The leader must go first, even through a door, to keep leading the group.

Sometimes we forget to make it easy to follow. "I was willing to do it the hard way, shouldn't others?" In a perfect world everyone would be as passionate about an issue as you are, but the truth is that they aren't. We get discouraged because other people aren't stepping up like you did. The truth is that if others aren't following, we haven't taken the time to understand the obstacles they see and help to eliminate them. Often the biggest obstacle is "I don't know how."

Getting five people to show up for a protest is easy. You plus four others who are equally passionate will show up.

What if you want 100 people to show up? It must be considerably easier for people who aren't as passionate to show up. You have to make it easy enough for people who aren't "passionate" but are just "concerned" to show up. They might need more detailed directions how to get there, where to park, how much walking will be involved. All those things are obvious to you but not to the larger group.

What if you want 1,000 people to show up? You still need to appeal to those who are passionate and concerned, but at this level you also have to make it easy to attend for someone who is just "curious." At this level, you need to have a celebrity or entertainer. That would give people two reasons to show up.

That's leadership. What, then, is technical leadership? Technical leadership is when someone paves the way to do things differently than they've been done before. It is when a person makes innovation happen.

Don't confuse technical leadership with "management." Management is a position on an org chart that specific people fulfill. Management is about setting priorities, providing resources, and removing roadblocks.

Technical leadership is different because it is something we can all do; in fact, it is something we all must do if our IT department is to be successful.

Technical leadership is about going first and making it easy for others to follow.

Technology is constantly changing and, therefore, an IT department must constantly be trying new things. For that to happen, someone must go first. Someone must be the first to try something. To transition to the new technology successfully, others must adopt it, too. To get others to adopt the new technology, you have to make it easy for people to adopt it who aren't as passionate as you.

For some of us, trying new things is easy. We're always trying new things!

Sometimes the problem is that there are too many things to try. Which to try first? We have to be selective. What are the three biggest problems in this department? What keeps you up at night? What does our boss complain about the most? Are there new products or services that will fix or alleviate those problems?

On the other hand, trying new things in an IT department is often a luxury. We're too busy to take a day to go through all the trouble to get the thing, learn the thing, evaluate it against other things, and demonstrate that this thing would be worth adopting in your department; however, if we invest the time required to try a new thing, we can save time for everyone else by sharing what we learned. More importantly, we can make it easy for others to follow in our footsteps. We do this by doing the groundwork, setting up the basic system, and finally creating a way that makes it easy for others to build on it.

Suppose your team is burdened with manually configuring machines. There are automated solutions out there but nobody has time to evaluate them all. Each evaluation means learning the system, deciding how it would fit into your environment, and so on. You take the time to evaluate a few, pick one, and set it up. Maybe it only maintains the configuration on three machines, and the configuration that it controls is modest: just keeping a few files in /etc properly configured. After you've done the hard work, it is time to make it easy for others to follow in your footsteps: you create a wiki page that explains how they can add new machines or start controlling additional aspects of the system. That's technical leadership.

Similar projects:

- Creating a repository for sysadmin scripts instead of having them scattered in people's home directories. You document how to add new scripts, update existing ones, and replicate the scripts to a new machine.
- Adopting a request ticket system instead of using email. You set it up with some basic categories and document for the rest of the team how to add/change/delete categories, FAQs, and get the most out of the system.
- Having a wiki for the team. You give a "brown bag" talk during lunch to teach people how to use it.
- Setting up a monitoring system so that you know what is broken before your users notice. Setting it up is difficult. Once that is done, configuring it to monitor new machines or services is easy. You document basic add/change processes with clear examples.

Technical leaders don't ask for permission. They may ask for feedback. They may ask for suggestions; however, people tend to dislike change. If you were to propose any of the above projects and ask, "Should I do it?" you'll probably be given 100 reasons why you shouldn't. On the other hand, if you do any of those projects and then give a demo about how it saves them time or improves their life, they will adopt it (especially if you've provided really good documentation: how to get started, how to add/change/delete items in the system, and so on).

The old-fashioned way to make yourself powerful within a company was to hoard information. You are the only person who knows how to do something, and if someone wants it done they must come to you and ask for your good graces. You hide information so that you can control it. You are the great and powerful

wizard that everyone must respect. That is the old way. The new way is to gain power by giving away information. You are the technical leader who set up the new repository, ticket system, wiki, or monitoring system. You went first and made it easy for others to follow. Now your power comes from teaching others to use that system. You are powerful because everyone in the organization remembers that you were the person who taught them how to do that thing and the other thing. You are powerful because your influence extends throughout the company because of all the people you've helped.

Technical leadership is something we can all do. For a modern company to survive, technical leadership is something we all must do. In fact, for the greater system administration community to survive we all must be technical leaders.

Professors, Campus Staff, and Students— do you have a USENIX Representative on your campus? If not, USENIX is interested in having one!

The USENIX Campus Rep Program is a network of representatives at campuses around the world who provide Association information to students, and encourage student involvement in USENIX. This is a volunteer program, for which USENIX is always looking for academics to participate. The program is designed for faculty who directly interact with students. We fund one representative from a campus at a time. In return for service as a campus representative, we offer a complimentary membership and other benefits.

A campus rep's responsibilities include:

- Maintaining a library (online and in print) of USENIX publications at your university for student use
- Distributing calls for papers and upcoming event brochures, and re-distributing informational emails from USENIX
- Encouraging students to apply for travel grants to conferences
- Providing students who wish to join USENIX with information and applications
- Helping students to submit research papers to relevant USENIX conferences
- Providing USENIX with feedback and suggestions on how the organization can better serve students

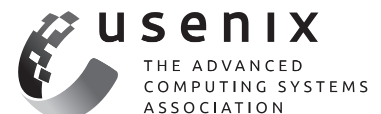
In return for being our "eyes and ears" on campus, representatives receive a complimentary membership in USENIX with all membership benefits (except voting rights), and a free conference registration once a year (after one full year of service as a campus rep).

To qualify as a campus representative, you must:

- Be full-time faculty or staff at a four year accredited university
- Have been a dues-paying member of USENIX for at least one full year in the past

For more information about our Student Programs, contact
Julie Miller, Marketing Communications Manager, julie@usenix.org

www.usenix.org/students



Log Filtering with Rsyslog

DAVID LANG

David Lang is a Staff IT Engineer at Intuit, where he has spent more than a decade working in the Security Department for the Banking Division. He was introduced to Linux in 1993 and has been making his living with Linux since 1996. He is an Amateur Extra Class Radio Operator and served on the communications staff of the Civil Air Patrol California Wing, where his duties included managing the statewide digital wireless network. He was awarded the 2012 Chuck Yerkes award for his participation on various open source mailing lists.
david@lang.hm

In my first *;login:* article [1], I provided an overview of how to build an enterprise-class logging system and recommended using rsyslog as the transport. For those who are not familiar with modern syslog daemons, this may seem like a strange recommendation. In this article I will provide an overview of rsyslog's capabilities, with the focus on its filtering capabilities. Where a traditional syslog limited you to filtering on the facility and severity reported by the application writing the logs, rsyslog lets you filter anything in the log message, as well as several things that are not.

Traditional Syslog

Traditional syslog messages have a facility value (the type of log it is) and a severity value (the importance of the message). These are combined to create the priority (PRI) of the message, which is a decimal number: PRI = Facility * 8 + Severity.

The log messages are sent between machines in the format:

```
<PRI>timestamp hostname syslogtag message
```

Normally, when the messages are written to a file, the <PRI> header is left off, so what shows up in the file is:

```
timestamp hostname syslogtag message
```

Syslog filters (located in /etc/syslog.conf) are in the form of:

```
facility.severity[,facility.severity] <whitespace> action
```

The possible actions are

- Write to a file
- Send to a named pipe
- Send to a remote machine via UDP
- Write to a terminal/console
- Send to users

The PRI value for a message is completely determined by the application that's creating the message, with no protection preventing any user from writing a message claiming to be from the kernel with a severity level of "emergency." This allows you to use some of the predefined system facilities for your application (say, news or UUCP), at the cost of confusing newcomers to your environment. Most people expect that all non-system applications are going to use one of the local* facilities.

An example /etc/syslog.conf file:

```
mail.*              /var/log/mail.log
auth,authpriv.*     /var/log/auth.log
*.*                 /var/log/messages
*.*                 @192.168.1.6
```

Almost every program that writes to syslog allows you to specify what facility to use, and almost none of them prevent you from configuring anything you want. With scripts, you can use the /usr/bin/logger command to log whatever you want.

```
$ logger -p kernel.emerg -t kernel -s "The system is on fire!!!"
```

results in a log that looks like the following and is tagged with the facility "kernel" and the severity "emergency":

```
Jul 21 19:55:43 myhostname kernel: The System is on fire!!!
```

Rsyslog

Rsyslog has a rapid development cycle compared to Linux distros. As of the time of writing, most Linux distros ship with rsyslog 5.x, while RHEL versions 6.3 and earlier ship with rsyslog 3.22 as the default. Rsyslog 5.x became an option in RHEL 5.9 and became the default in RHEL 6.4. Meanwhile, the current supported version is rsyslog 7.4. As can be expected, the current upstream versions include many features that are not available in older versions. Adiscon, the primary sponsor of rsyslog, development hosts repositories for the newest versions for both RHEL/CentOS and Ubuntu packages, and several other people maintain current packages for other systems [2].

Among the many changes in rsyslog 6.x there was a new config syntax added. Unless stated otherwise, all examples provided in this article have been tested with rsyslog 3.x or newer.

Rsyslog has a modular design and, in addition to the capabilities of traditional syslog, supports many other modules that offer many additional functions.

Input Modules accept input into rsyslog:

```
im3195, imdiag, imfile, imgssapi, imjournal, imklog, imkmsg,
immark, impstats, imptcp, imrelp, imsolaris, imtcp, imttcp, imudp.
imuxsock, imzmq3
```

Stackable Parser Modules parse or modify the data the input modules accepted:

```
pmrfc3164, pmrfc5424, pmaixforwardedfrom, pmcisconames,
pmlastmsg, pmrfc3164sd, pmsnare
```

Message Modification Modules modify the parsed message or create variables from the message:

```
mmanon, mmaudit, mmcount, mmfields, mmjsonparse, mmnormalize,
mmsnmptrapd
```

Output Modules deliver the message to a destination:

```
omelasticsearch, omgssapi, omhdfs, omhiredis, omjournal,
omlibdbi, ommail, ommongodb, ommysql, omoracle, ompgsql,
omprog, omrabbitmq, omrelp, omruleset, omsnmp, omstdout,
omtesting, omudpspoof, omuxsock, omzmq3, omfwd (tcp/udp
network delivery), omdiscard, omfile, ompipe, omshell, omusrmsg
```

String Generation Modules provide predefined templates such as the following built-in templates:

```
RSYSLOG_DebugFormat, RSYSLOG_FileFormat,
RSYSLOG_ForwardFormat, RSYSLOG_SysklogdFileFormat,
RSYSLOG_SyslogProtocol23Format,
RSYSLOG_TraditionalFileFormat,
RSYSLOG_TraditionalForwardFormat
```

Compatibility with Traditional Syslog

Rsyslog supports the traditional PRI-based filtering syntax, so if your current usage fits within this syntax, you can continue to use it.

At startup, rsyslog needs a little more information in its config file to tell it which input modules to load and how to configure them, but the filtering lines can be identical.

An example /etc/rsyslog.conf equivalent to the /etc/syslog.conf shown earlier would be:

```
$ModLoad imuxsock
$ModLoad imklog
$ModLoad imudp
$UDPServerRun 514
mail.*                  /var/log/mail.log
auth,authpriv.*         /var/log/auth.log
*.*                     /var/log/messages
*.*                     @192.168.1.6
```

Because rsyslog has an include function, the /etc/rsyslog.conf could be simplified to:

```
$ModLoad imuxsock
$ModLoad imklog
$ModLoad imudp
$UDPServerRun 514
$IncludeConfig /etc/rsyslog.conf
```

Several Linux distros use the line:

```
$IncludeConfig /etc/rsyslog.d/*.conf
```

to let you manage the configurations for different applications in separate application-specific files, without having all configuration information collected in the same file. There is a bug in rsyslog 6.x and 7.0 (fixed in 7.2) that caused the included files to be processed in reverse order. One caution with included files: rsyslog includes all the files and then evaluates the resulting config. This means that if you set a configuration value in one included file, it will still be in effect for later included files.

Modification of the Outbound Message

Rsyslog also allows you to change the log message that it sends out to any destination. You can create a format template [3] with a config entry like:

```
$template strangelog,"text %hostname:::lowercase% %msg% more\n"
```

where the items between '%' are variables (with formatting options).

Then in your action, you can tell rsyslog to use a specific template instead of the default template:

```
*.*     /var/log/messages;strangelog
```

Rsyslog even lets you create a template for the filename, so you could use a configuration like:

```
$template sortedlogs="/var/log/messages-%fromhost-ip%"
*.*     ?sortedlogs;strangelog
```

to write the messages to different files, with filenames in the format specified by the "sortedlogs" template, based on the source IP address.

Variables Available

Rsyslog provides different flavors of variables for use in config files: message property variables, trusted property variables, message content variables, and user-defined variables.

Message Property Variables

These are items derived from the message or the connection information, such as the timestamp within the message, the timestamp when the message was received on the local system, the hostname in the message, the hostname/IP of the system that delivered the message to the local box, PRI info, etc. [4]. For rsyslog version 5 and earlier, these were the only variables available.

Trusted Properties

Late in the 5.x series, rsyslog implemented the ability to query the kernel to get information about the process on the other end of the /dev/log socket (UID, PID, name of binary, command line, etc.), so that it could log information that normal non-root user processes cannot forge (processes running as root can still forge this information). In rsyslog 5.x, the information could only be appended to the log message, but with 6.x and newer, this information can be turned into variables.

Variables Parsed from Message Content

Rsyslog version 6 introduced "message modification" modules. These modules are allowed to modify a message after it has been parsed, and they can be invoked as the result of a filter test. In addition to modifying the message, these modules can also set variables that can be used the same way that the properties defined above are used.

The two most significant message modification modules for creating variables are mmjsonparse and mmnormalize.

Mmjsonparse will parse a JSON-formatted message and create a tree of variables for you to use. This was implemented to support the CEE logging standard, and requires that the JSON start with *@cee:*.

These rsyslog.conf additions are needed to use this module:

```
$ModLoad mmjsonparse
*.* :mmjsonparse:
```

This supports multiple levels of structure: `$!root!level1!level2!` etc. refers to an individual item, $! refers to the entire tree, and `$!root!level1` refers to a partial subtree.

Mmnormalize [5] lets you define a rule set for parsing messages, and it will do a very efficient parse of the log message, creating variables.

For example, starting with this example log message:

```
Jul 21 19:55:03 kernel: [1084540.211910] Denied: IN=eth0 OUT=
MAC=00:30:48:90:cc:a6:00:30:48:da:48:e8:08:00 SRC=10.10.10.10
DST=10.10.10.11 LEN=60 TOS=0x10 PREC=0x00 TTL=64 ID=28843
DF PROTO=TCP SPT=44075 DPT=444 WINDOW=14600 RES=0x00 SYN
URGP=0
```

and the rule file normalize.rb:

```
rule=: %kerntime:word% Denied: IN=%in:word% OUT=
MAC=%mac:word% SRC=%src-ip:ipv4% DST=%dst-ip:ipv4%
LEN=%len:number% TOS=%tos:word% PREC=%prec:word%
TTL=%ttl:number% ID=%id:number% %DF:word% PROTO=%proto:word%
SPT=%src-port:number% DPT=%dst-port:number%
WINDOW=%window:number% RES=%res:word% %pkt-type:word%
```

produces this log message:

```
Jul 21 19:55:49 myhostname json_msg: @cee:{ "urgp": "0",
"pkt-type": "SYN", "res": "0x00", "window": "14600", "dst-
port": "444", "src-port": "51954", "proto": "TCP", "DF": "DF",
"id":"31890", "ttl": "64", "prec": "0x00", "tos": "0x10", "len":
"60", "dst-ip":"10.10.10.10", "src-ip": "10.10.10.11", "mac":
"00:30:48:90:cc:a6:00:30:48:da:48:e8:08:00", "in": "eth0",
"kerntime":"[1152127.460873]" }
```

You do need to add the following lines to rsyslog.conf to use this module:

```
$ModLoad mmnormalize
$mmnormalizeUseRawMSG off
$mmnormalizeRuleBase /rsyslog/rulebase.rb
*.* :mmnormalize:
$template json_fmt,"%timereported% %hostname% json_msg: @
cee:%$!%\n"
*.*   /var/log/test;json_fmt
```

User-Defined Variables

Rsyslog versions 7 and later allow you to define your own variables in the config file in addition to the ones created by the message modification modules. In rsyslog 7.6 there will be three flavors of variables that you can create:

- **Normal variables,** which can be created by "message modification modules" or by config statements. These are addressed as "$!name".
- **Local variables,** which cannot be set by message modification modules. These are addressed as "$.name".
- **Global variables,** which cannot be set by message modification modules, but will persist across log messages (other variables are cleared after every message is processed). These are addressed as "$/name".

Here are some examples of defining variables. Unlike other config statements, set and unset require a trailing ';':

```
set $!user!level1!var1="test";
set $!user!level1!var1=$!something + 1;
unset $!user!level1;
```

Using Variables

One common problem that people run into when using variables is the fact that the different types of variables were added to rsyslog at different times, and as a result there are different ways they are named.

The traditional message property variables have just the variable name, such as "timereported" or "fromhost-ip".

Other properties, mostly referring to the runtime environment (rather than the log message), have names like "$myhostname" or "$now":

- Variables parsed from the message with mm modules have names like "$!name".
- Local variables have names like "$.name".
- Global variables have names like "$/name".
- When using variables, the examples usually have the classic properties, so you see things like:
- %msg% in a template
- :msg, in a property-based filter
- $msg in a script-style config

But when you are using the other variable types, you must be aware that the variable prefix ('$' '$!' '$.' '$/') is considered part of the variable name, not a reference to it, so you would use something like:

- * %$!portnumber% in a template
- * :$!portnumber in a property-based filter

But you only use "$!portnumber" not "$$!portnumber" in a script-style filter or new-style config statement.

You can use "$$!portnumber" without syntax errors in some cases, but this results in an indirect reference to something.

New Filtering Capabilities

Use Last Match

The simplest and fastest "filter" to use is the '&' filter; It isn't really a filter because it just tells rsyslog to use the result of the last test. If that last test matched, the '&' will match as well.

This is extremely useful for cases in which you want to do several things if a condition is met.

A common example is when you want to log all messages of a particular type in one place, and send them off to another system.

```
mail.*    /var/log/mail.log
&         @mailanalysis
```

With rsyslog version 6 and later you can use {} to group multiple actions together, and as a result '&' isn't needed as much as it used to be.

```
Mail.*    { /var/log/mail.log
          @mailanalysis }
```

Stop Processing This Log Action

When you know that you don't want to process a log message any longer, you can tell rsyslog to stop and not waste time checking any further rules. This is commonly used in conjunction with the & filter or a block of actions to prevent rsyslog from trying to match any other filter rules after you have done what you want with a message. Without a stop, the message will get sent to every output that has a matching filter:

```
mail.*    /var/log/mail.log
&         @mailanalysis
&         ~
```

Or with the rsyslog version 6+

```
Mail.*    { /var/log/mail.log
        @mailanalysis
stop }
```

and rsyslog will stop processing this message and no other rules will be checked. Be careful—using included config files as a stop in one file may have an unexpected impact on the processing of another file.

"Always" Filter

In rsyslog version 7, the config optimizer is able to identify actions that have no filter in front of them. So instead of writing lines like:

```
*.*  /var/log/messages
*.*  @loghost
```

you can just write:

```
/var/log/messages
@loghost
```

The optimizer will optimize away "always match" filters in any case, so there is no performance penalty to continuing to write things the traditional way.

Property-Based filters

rsyslog has long supported property-based filters [6], which are formatted as:

```
:variable, [!]compare-operation, "value"
```

Examples of the different types are:

```
:programname, isequal, "sendmail" /var/log/mail.log
:msg, contains, "(root) CMD)"    ~
:msg, startswith, "pam_unix"     /var/log/auth.log
```

With property-based filters, you are no longer limited to filtering on the PRI value that was defined in the message. You can now filter based on the program name, or anything else in the log message. As a result, seeing rsyslog config files that have few (if any) PRI-based filters is common, and even those tend to be *.* or *.severity type filters, completely ignoring the facility.

For example, to file different types of logs into different output files, the following type of config is common:

```
:programname, startswith, "%ASA"    /var/log/cisco-messages
& ~
:programname, startswith, "postfix" /var/log/postfix-messages
& ~
:programname, isequal,    "snmpd"   /var/log/snmpd-messages
& ~
:programname, isequal,  "sendmail" /var/log/sendmail-messages
& ~
```

Script-Based Filters

Property filters can only test one thing, so rsyslog also includes script-based filters. These are familiar looking if-then conditions. Prior to the config optimizer that was added in rsyslog version 7, these were slow compared to PRI filters and significantly slower than property-based filters. In rsyslog version 7, the optimizer makes all the different formats equivalent in speed.

Script-based filters look like [7]:

```
IF test THEN action [ELSE action]
```

where test can be an arbitrarily complex expression, with normal precedence of operations, Boolean short-cutting, and built-in functions.

Action can be just about any block of config statements (including nested IF statements). Not all config items can be put into the "then" section of a test. In general, setup type commands (template definitions, input definitions, config parameters that change rsyslog internals) are not allowed. Commands that do some sort of action (set a variable, send the message to an output, invoke message modification modules) are allowed.

The equivalent to the property filter example would be:

```
if $programname startswith("%ASA") then /var/log/cisco-messages
else if $programname startswith("postfix") then
   /var/log/postfix-messages
else if $programname startswith("snmpd") then
   /var/log/snmpd-messages
else if $programname startswith("sendmail") then
   /var/log/sendmail-messages
else {
   <rest of rules>
}
```

Array Matches

Starting in rsyslog 7.2, repeated similar tests can be greatly optimized with "array" matches. Rather than having tests for many possible matches formatted like:

```
if $programname == "postfix" or $programname=="exim"
    or $programname=="sendmail" then /var/log/mail.log
```

rsyslog now supports what it calls Array Matches.

This allows you to write the test as:

```
if $programname == ["postfix","exim","sendmail"] then
   /var/log/mail.log
```

This can be extremely powerful when you combine it with dynamic file templates:

```
$template maillogs,"/var/log/mail-%programname-%severity%"
if $programname == ["postfix","exim","sendmail"] then ?maillogs
```

This will split the log files for mail apps into separate files for each type of program and severity level.

New Config Syntax

The old syntax will continue to be supported, and you can freely mix and match between the different config syntaxes within the same file (or included files) so you don't have to change your config files when you upgrade. For some of the newer functionality, though, you must use the new syntax.

In the old config syntax, you must set up options and then execute the function, while the new format looks like function calls with many parameters.

This example is in the old syntax:

```
$mmnormalizeUseRawMSG off
$mmnormalizeRuleBase /rsyslog/rulebase.rb
*.* :mmnormalize:
```

Here is the equivalent config using the new syntax:

```
action(type="mmnormalize" UseRawMsg="off"
   ruleBase="/etc/rsyslog.d/normalize.rb")
```

Here's another example using the old syntax for a more complex action (sending a SNMP trap):

```
$actionsnmptransport udp
$actionsnmptarget 192.168.1.100
$actionsnmptargetport 162
$actionsnmpversion 1
$actionsnmpcommunity testtest
$actionsnmptrapoid 1.3.6.1.4.1.19406.1.2.1
$actionsnmpsyslogmessageoid 1.3.6.1.4.1.19406.1.1.2.1
$actionsnmpenterpriseoid 1.3.6.1.4.1.3.1.1
$actionsnmptraptype 2
$actionsnmpspecifictype 0
*.* :omsnmp:
```

With the new syntax, the same config appears in a much more compact format:

```
action(type="omsnmp" transport="udp" server="192.168.1.1"
   trapoid="1.3.6.1.4.1.19406.1.2.1" port="162" version="1"
   messageoid="1.3.6.1.4.1.19406.1.1.2.1" community="testtest"
   enterpriseoid="1.3.6.1.4.1.3.1.1" traptype="2" specifictype="0")
```

On the other hand, some things are simpler with the old config.

```
$template strange,"some text %variable% %variable:modifiers%\n"
```

is significantly longer using the new syntax:

```
template(name="strange" type=string
   string="some text %variable% %variable:modifiers%\n")
```

Even this simple rule in the old syntax:

```
*.*     /var/log/messages;templatename
```

becomes longer, although a bit more obvious as to what it does, using the new syntax:

```
*.*     action(type="omfile" File="/var/log/messages"
          Template="templatename")
```

Choosing which syntax to use is completely up to you— use whichever you find easier for the task at hand. Most configurations will include a mix of old and new, but in general, the more complex the configuration, the more likely you are to benefit from the new config syntax. Prior to the config optimizer added in rsyslog v7, PRI-based filters were by far the fastest type of filter to use.

Example

In the first article, I recommended using recent versions of rsyslog on the Aggregator and Analysis farm machines, so that you can take advantage of the greatly expanded capabilities and performance of the newer versions. One of the recommendations that I made was to use JSON-structured messages for the transport so that additional metadata could be added. The following config files are an example of what you may want to use.

Note that in these examples, I mix old and new config styles and make use of the "always" filter.

On all "normal" systems (app-servers, routers, switches, etc.), deliver all messages to the Edge Aggregation servers. On *nix systems, add an entry like the following to /etc/syslog.conf or /etc/rsyslog.conf:

```
*.* @edge-server-for-local-network
```

Here is an example /etc/rsyslog.conf for an Edge Aggregator. Note that rsyslog treats newlines as whitespace, so no line continuation characters are necessary. The exception to this is the $template command, which needs to be on one line (but is split here for printing):

```
module(load="imuxsock" SysSock.Annotate="on"
    SysSock.ParseTrusted="on")
module(load="imklog")
module(load="imudp")
input(type="imudp" port="514")
module(load="imtcp" MaxSessions="1000")
input(type="imtcp" port="514")
module(load="mmjsonparse")
action(type="mmjsonparse")
if $fromhost-ip != "127.0.0.1" then {
   # if the log is being received from another machine,
   #    add metadata to the log
   set $!trusted!origserver = $fromhost-ip;
   set $!trusted!edge!time = $timegenerated;
   set $!trusted!edge!relay = $$myhostname;
   set $!trusted!edge!input = $inputname;
```

```
} else {
   set $!trusted!local!input = $inputname;
}
# set this to reflect the environment that this Edge server is
servicing
set $!trusted!environment = "Dev network";
# note the template must be on a single line
# wrapping is for display only
$template structured_forwarding,
   "<%pri%>%timereported% %hostname% %syslogtag% @cee:%$!%\n"
/var/log/messages;structured_forwarding
@@core-server
#send to the core server via TCP consider using RELP instead
```

And here is an example configuration for the Analysis Farm systems:

```
module(load="imuxsock" SysSock.Annotate="on"
   SysSock.ParseTrusted="on")
module(load="imklog")
module(load="imtcp" MaxSessions="1000")
input(type="imtcp" port="514")
module(load="mmjsonparse")
action(type="mmjsonparse")
if $fromhost-ip == "127.0.0.1" then {
   #if this is a local log, send it to an edge relay.
   set $!trusted!local!input = $inputname;
   @edge-server
   stop }
$template std,"%timereported% %hostname% %syslogtag%%$!msg%\n"
/var/log/messages;std
```

To demonstrate how this works, on an app-server I executed:

```
logger testtest
```

which produced this message in /var/log/messages:

```
Jul 24 14:51:42 app-server dlang: testtest
```

On the Edge Aggregator server, the log message is reformatted and metadata is added to produce the following log entry that is sent to the Core Aggregator (which then relays the message to all Analysis Farms):

```
<13>Jul 24 14:51:42 app-server dlang: @cee:{ "msg": "testtest",
"trusted": { "origserver": "10.1.2.9", "edge": { "time":
"Jul 24 21:51:42", "relay": "edge-server", "input": "imudp" },
"environment": "Dev network" } }
```

Note that with the app-server set to Pacific time and the edge server set to GMT, the timestamp when the log was created doesn't match when it's received.

And, finally, on the Analysis Farm systems, the following message will be produced in /var/log/messages:

```
Jul 24 14:51:42 app-server dlang: testtest
```

This threw away all the metadata, resulting in a message that looks identical to what was originally generated, but the metadata was available for filtering decisions up to this point. And a slightly different format on the Analysis Farm server could make any of the metadata available to the analysis tools.

As a second demonstration, on an Edge Aggregator I again executed:

```
logger testtest
```

Because this adds trusted properties to the message, it sends the following log entry to the Core Aggregator:

```
<13>Jul 24 21:53:39 edge-server dlang: { "pid": 4346, "uid":
1000, "gid": 1000, "appname": "logger", "cmd": "", "msg":
"testtest", "trusted": { "local": { "input": "imuxsock" },
"environment": "sending network" } }
```

Again, the Analysis Farm server will throw away the extra metadata and reformat the log to be:

```
Jul 24 21:53:39 edge-server dlang: testtest
```

But this time there was more metadata available about the process that created the log message available prior to the final output.

In conclusion, rsyslog has tremendous flexibility in processing your log messages. You can filter on just about anything that you care about, and you can modify messages as you send them out to any of the many different supported outputs.

References

[1] David Lang, "Enterprise Logging," *;login:*, vol. 38, no. 4: https://www.usenix.org/publications/login/august-2013-volume-38-number-4/enterprise-logging.

[2] http://www.rsyslog.com/doc/rsyslog_packages.html.

[3] http://www.rsyslog.com/doc/rsyslog_conf_templates.html.

[4] http://www.rsyslog.com/doc/property_replacer.html.

[5] http://www.rsyslog.com/doc/mmnormalize.html.

[6] http://www.rsyslog.com/doc/rsyslog_conf_filter.html.

[7] http://www.rsyslog.com/doc/rainerscript.html.

Flash Caching on the Storage Client

DAVID A. HOLLAND, ELAINE ANGELINO, GIDEON WALD, AND MARGO I. SELTZER

David A. Holland is a researcher in software systems at Harvard. His core research interest is figuring out how to write better software, which covers a broad range of projects and applications. He wrote the OS/161 instructional operating system. dholland@eecs.harvard.edu

Elaine Angelino is a Ph.D. student in Computer Science at Harvard SEAS. Her advisor is Professor Margo Seltzer. elaine@eecs.harvard.edu

Gideon Wald is an entrepreneur living and working in San Francisco. He was a product manager at Google for three years on Search and Chrome before leaving to co-found a nascent company in the enterprise software space. gideon.wald@gmail.com

Margo Seltzer is the Herchel Smith Professor of Computer Science at Harvard's School of Engineering and Applied Sciences, an architect at Oracle Corporation, and the current USENIX board President. Her research and commercial activities revolve around all sorts of systems: operating systems, database systems, file systems, learning systems, etc. margo@eecs.harvard.edu

Most use of flash memory for caching so far has been on the storage server side. Using a trace-driven simulator we examined the use of flash as a large client-side cache. We found that the benefit of such a cache derives chiefly from its size, not the persistence of flash; but persistent caches offer additional benefits. We also found that the cache can be write-through without harming performance, and that for some workloads it allows freeing up system RAM that would otherwise be needed for caching.

In recent years, flash memory has gained attention not only as a medium for storage but also as a component of storage system caches. Most such uses have been on the server side: flash deployed in direct combination with disks. Our study [1] examined the use of flash on the client side of a network, such as on the compute nodes in a cluster. This arrangement reduces access latency and network load at the cost of requiring a flash device on each node. For shared storage, it can also introduce cache consistency problems. We ran simulations to examine the range of possible designs of this type and their various costs and benefits.

In our system model (Figure 1), an application performs I/O into a RAM cache (the ordinary operating system disk cache), which connects in turn to a flash cache. These components access a file server across a network. Many scenarios, ranging from Web application servers to render-farm nodes, share this basic structure.

We treat the flash cache as a SATA-attached solid-state drive. PCI flash devices that behave like SATA-attached drives should give similar results. We also modeled the file server as a "smart" enterprise-grade filer with lots of fancy prefetching and caching logic. The flash cache will help plain disk arrays more as they are slower.

Design Space

We examined the tradeoffs that arise when designing a client-side flash cache. We asked four key questions: whether the flash cache can/should be write-through or write-back, the degree of integration with the operating system required, the cost/benefit of cache persistence, and the need for cache consistency management.

The motivating question for this study was whether the flash cache can be write-through. With a write-through cache, managing crash recovery and maintaining cache consistency is easier; however, write-back caches generally perform better. We wanted to know the magnitude of this effect.

Another question was whether the flash cache must be integrated with the operating system and the operating system's disk cache. An implementation that operates as an independent layer will be much easier to build and deploy; however, an integrated implementation might potentially perform much better, so we wanted to know what the tradeoffs would be.

The third question was whether the flash cache needs to survive crashes. A persistent cache must store recoverable cache metadata in the flash, as opposed to just using RAM; this creates additional overhead. On the other hand, as (re)filling a 64 GB cache to full effectiveness can take hours or even days, not making the cache persistent can lead to substantial periods

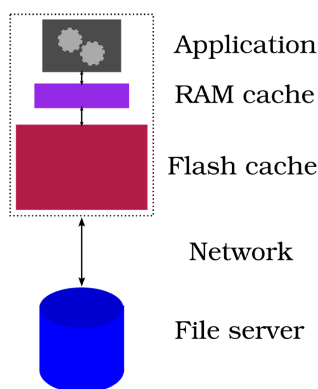


Figure 1: System model

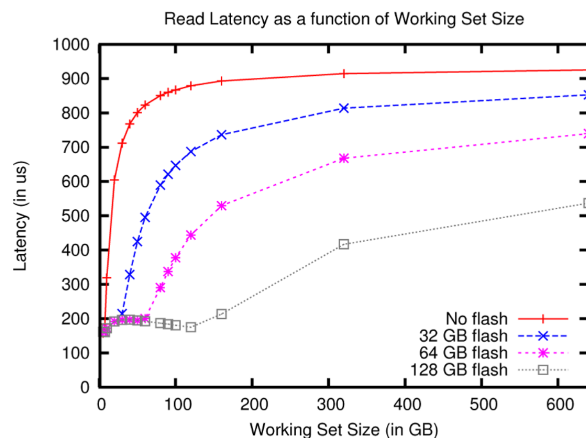


Figure 2: Application read latency as a function of working set size

of reduced performance. We wanted to find out how much the performance would be reduced, and for roughly how long.

Finally, we wanted to know what the consequences would be for cache consistency management. We were chiefly concerned with serving private disk images, but shared storage volumes are also important.

As the design space produced from these questions is enormous, we chose to do a simulator-based study that would allow us to explore these tradeoffs relatively inexpensively.

Our simulator reads a trace of I/O events, where each event is a read or write access to a particular region of a file, done by a specific thread on one of perhaps many hosts. The traces we used for our study were statistically generated using a tool we wrote for the purpose. We did use some real traces to validate the simulator against an existing implementation (NetApp's Mercury); this allowed us to be reasonably confident that the simulator was producing plausible results.

Results

Our first result was not the answer to a design question but a rather more basic issue: whether a client-side flash cache is a win. It is; a client-side flash cache provides a fairly substantial benefit, both for medium-sized workloads that fit into the flash but do not fit into RAM and for large workloads that do not fit into even a large flash device.

Figure 2 shows the average latency seen by the application for read operations (per 4096-byte block) for a range of workload sizes and four different flash sizes. This is with an 8 GB RAM cache; the workloads are 30% writes and 70% reads. At the bottom left where the workload fits into the cache, a large flash cache offers in-cache performance for much larger workloads than possible without it; on the right where the workload is 5x to 10x the flash size there is still a substantial benefit.

In this environment the file server's prefetching performance is critical. The application's read latency is dominated by reads that have to go all the way to disk. (This takes milliseconds and everything else is measured in microseconds.) If—by inserting a large cache in front of the filer—we hamper the filer's ability to prefetch, we can easily lose most or all of the flash cache's performance gain. We believe that adjusting the filer's internal tuning can avoid this effect; however, deploying client-side flash caches in front of an old filer that does not know how to cope may not provide the benefit that one might expect.

The flip side of this issue is that the ability of a plain disk array to prefetch is negligible under all circumstances compared to a filer. So when the backend is a plain disk array, the flash cache offers a much greater benefit.

Our first design question above was whether the flash cache could be write-through or whether this hampers performance. Also, the RAM cache needs to write data back to the flash cache; policies that work well for disks might not be appropriate in this environment. To investigate this we implemented four simple cache write-back policies:

- Synchronous write-through: block the app until the write to the next layer is complete.
- Asynchronous write-through: start writing to the next layer immediately, but do not block on it.
- Periodic: every so often a background thread writes out modified blocks.
- None: let the cache fill and write updates back only when evicting old blocks.

Trying four different time periods for the periodic policy gives seven settings each for the RAM and flash caches, making forty-nine cases in total.

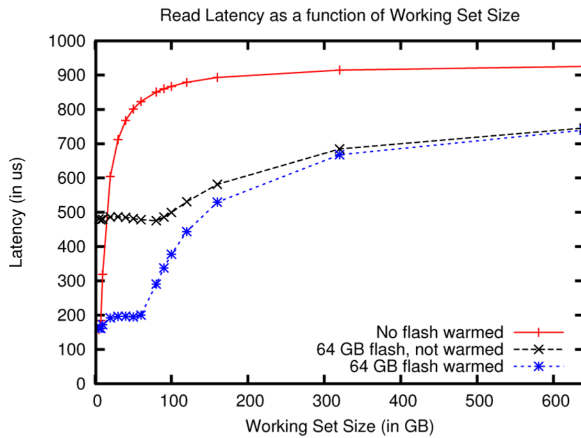


Figure 3: Effect of persistence on application read latency

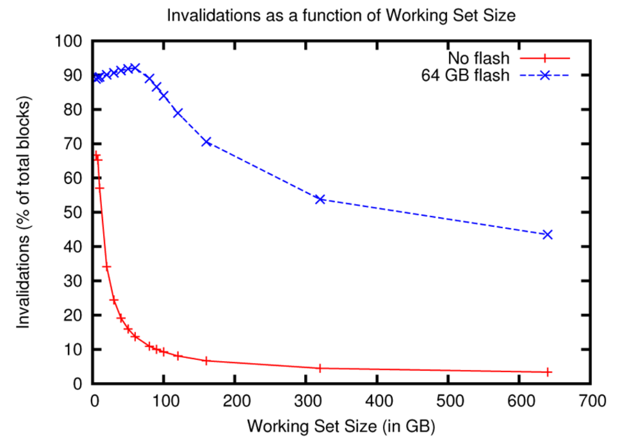


Figure 4: Invalidations required as a function of working set size

The obviously silly policies, such as synchronously writing from the RAM cache while the application waits, perform badly. ("None" turns into "synchronous" once the cache fills and also performs badly.) Otherwise, we found (somewhat to our surprise) that all other policies perform identically. The flash is large enough that as long as changes get written in some reasonable way, there is plenty of room for new incoming data.

Consequently, we did not try anything more complicated. The conclusion is that the flash cache can be write-through without hurting performance. This makes dealing with cache consistency for shared volumes much easier.

The second design question we addressed was whether the flash cache needs to be integrated with the operating system buffer cache. We compared the "naive architecture," in which the flash appears as an independent layer underneath the RAM cache with no integration whatsoever, and the "unified architecture," where the flash and RAM are fully integrated into a single cache framework. We found that the unified cache performed better for reads and worse for writes.

The chief difference between these models is that in the naive architecture the contents of the RAM cache become duplicated in the flash. The unified cache can avoid this and as a result becomes effectively larger. By tinkering with timings and settings, we ascertained that the improved read performance of the unified cache was exactly due to this effect. Given the price of flash compared to the assorted costs of implementing and deploying a unified cache, buying more flash is much cheaper.

Meanwhile, the worse write performance arose from an implementation issue: writes go to the next available block. With 8 GB of RAM and 64 GB of flash, 8/9 of the blocks are flash; the average write latency seen by the application was 8/9 of the flash write latency. A smarter implementation could hide this latency.

Persistence

Much of the benefit of using flash for caching comes simply from its size and speed; however, because flash is persistent, an obvious question is whether the flash cache should be persistent as well. As discussed earlier, this has both benefits and costs.

To approximate the performance overhead, we doubled the simulated time for writing to the flash: one write for the data and another write for metadata. This is pessimistic: in practice one can get away with much less metadata write traffic. There was no visible effect whatsoever on the application: given a reasonable policy for writing from the RAM cache to the flash cache, these writes happen in the context of the kernel's background processes and are fully hidden from the application.

To investigate the benefit, we ran the same workloads on warmed and unwarmed caches. Normally we use the first half of each generated I/O trace to warm up the cache and collect timing data on the second half. For the unwarmed case, which is equivalent to crashing right before starting the workload, we skipped the first half instead.

The results are shown in Figure 3. This graph requires some explanation. It shows application read latency for three cases: no flash cache, an unwarmed flash cache, and a warmed flash cache. In our study we pegged the total run size of our traces to the working set size; each trace pushes through a volume of twice the working set size during the measurement phase. Therefore, for the smallest workloads (left side of the graph) the trace finishes long before the flash cache fills, and the behavior shown on the graph is the performance seen during the warming phase. Moving to the right, the traces become far larger than a 64 GB flash, and the average behavior over the whole trace converges to the behavior with a warm cache.

What this shows is that the performance with a cold cache is considerably worse than with a warm cache, but the cache

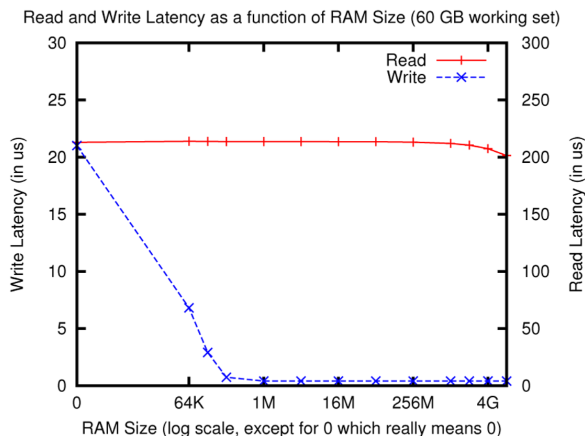


Figure 5: Application read and write latencies with small RAM sizes and 60 GB working set

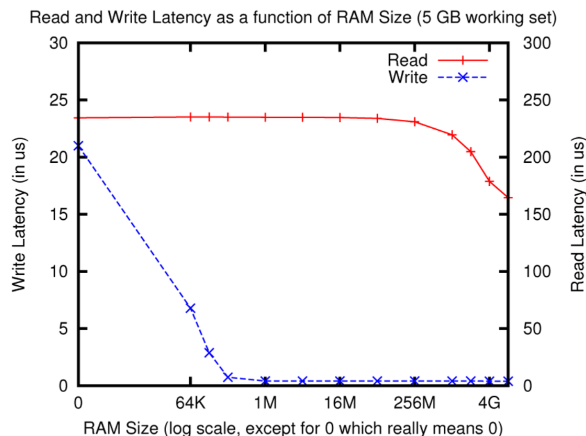


Figure 6: Application read and write latencies with small RAM sizes and 5 GB working set

warms rapidly enough that having it is still better for all but the very shortest and smallest workloads. In simulator time, the smallest workload in this graph completed in less than ten minutes; the largest took about a day. The cross-over point between the no-flash and cold-flash lines corresponds to roughly 20–25 minutes. How simulator time corresponds to real time in real-life workloads is not so clear. Twenty minutes of simulator time might correspond to several hours of real time, depending on the intensity and concurrency of the workload.

The conclusion, however, is that while making the cache persistent offers significant and noticeable performance gains, unless you plan to be crashing regularly it isn't necessary to realize much of the cache's performance gain.

Cache Consistency

As mentioned above we were primarily looking at serving private disk images; however, shared data is also important and cache consistency is a significant issue when handling it. This is a complex problem with complex solutions; we did not implement any particular cache consistency protocol in our simulator. Instead we used a simple scheme where the simulator took advantage of its own global knowledge to automatically invalidate stale blocks wherever they appeared. The results we have, therefore, do not take into account the network traffic generated by a cache consistency protocol; but they do take into account the overhead caused by needing to re-fetch blocks that have become obsolete.

Figure 4 shows the percentage of writes that incurred an invalidation over a range of working set sizes. This is for two hosts sharing the same working set (a fairly adverse situation); as elsewhere, this is with an 8 GB RAM cache and 30% of the I/Os are writes.

For workloads that fit into the flash cache, upwards of 90% of write operations cause an invalidation. This is much higher than without the flash, even for the smallest workloads that fit into RAM. And for larger workloads, the invalidation rate drops off much more slowly.

This effect is potentially enough to affect the performance or scalability of existing cache consistency protocols. An additional problem arises for persistent caches of shared data: a host that is offline and rebooting cannot participate in an online cache consistency protocol and would need to be able to catch up afterwards.

Our study and our materials do not really examine consistency issues in detail; further work, including a detailed implementation of one or more specific protocols, is probably indicated. But we can tentatively conclude that with shared data, particularly broadly shared data and particularly for write-heavy workloads, consistency management overhead may erase most or all of the benefit of the client-side cache.

No RAM Cache

We came across an additional unexpected phenomenon: in at least some cases, it appears that cutting back the amount of RAM used for caching to (almost) zero makes sense. Figure 5 shows the read and write latency seen by the application as the RAM size is reduced (moving right to left) from the default 8 GB down to 64 KB and then all the way to zero. For all points the flash size is 64 GB; the RAM-to-flash writeback policy has been changed to asynchronous write-through.

Notice that the write latency remains the same all the way down to 256 KB of RAM . . . and the read latency is effectively unchanged. The read latency is slightly worse compared to the largest RAM sizes, but this effect is negligible (around 2%).

Upon reflection one might expect this result, because the working set is much larger than the RAM size and the hit rate in the RAM cache is miserably low. (With 8 GB of RAM the hit rate is about 14%; the flash hit rate is over 85%.) The effect appears to a surprising extent even in small workloads. Figure 6 shows the same thing, but for a 5 GB workload. The far right point is for 8 GB of RAM, in which the working set fits completely. The penalty here is about 25–30%. This is substantial, but it is not necessarily fatal. There are almost certainly workloads where a 30% reduction in read performance is worth being able to repurpose 8 GB of RAM; for example, there are many applications where an extra 8 GB will more than offset this penalty.

This tradeoff is made possible by the flash cache; without the flash, the cost of shrinking the RAM cache is not merely 25–30%; reads become some *five times* slower.

One of the less obvious reasons for this effect is that in our workloads, like most real workloads, some accessed data is outside the working set. These I/Os tend to miss in normal-sized caches; the flash is large enough to help with them.

We should also stress that this is something of a preliminary result, in that we are not yet sure how well it will translate to real-life workloads in real-life situations. But it certainly bears consideration.

Conclusions

The results of our simulations show that even the simplest form of client-side flash caching provides significant benefits to applications. We also identified a number of points that simplify the space of designs worth pursuing. First, it is perfectly fine from a performance standpoint for the flash cache to be write-through, or to use any other reasonable write-back policy. Second, there is no need to integrate the flash cache tightly with the operating system; the benefit of doing so is purely that the cache becomes slightly larger, but it is much cheaper to buy more flash. Third, much of the benefit of the flash cache can be gained without making it persistent; however, persistence offers additional benefits, incurs little or no overhead in practice, and is probably worthwhile. Fourth, cache consistency becomes a serious issue with caches of this size if multiple hosts are actively modifying overlapping working sets. Even with a write-through cache, such workloads cause substantially more invalidation traffic than we see with traditional RAM-based caches. Traditional cache consistency protocols may also not be able to cope with a persistent cache being offline during a reboot.

Acknowledgments

This work was supported by NetApp. Additionally, James Lentini, Keith Smith, and Chris Small, all of NetApp, were tremendously helpful in providing us with the means and expertise to validate our simulator.

References

[1] D. A. Holland et al., "Flash Caching on the Storage Client," Proceedings of the 2013 USENIX Annual Technical Conference (San Jose, CA, 2013).

NOVEMBER 3–8, 2013 • WASHINGTON, D.C.

Lucky LISA '13

27th Large Installation System Administration Conference

Keynote Address: "Modern Infrastructure: The Convergence of Network, Compute, and Data" by Jason Hoffman, *Founder, Joyent*

Join us for **6 days of practical training** on topics including:

- **SRE Classroom: Non-Abstract Large System Design for Sysadmins** by John Looney, *Google*
- **Root Cause Analysis** by Stuart Kendrick, *Fred Hutchinson Cancer Research Center*
- **PowerShell Fundamentals** by Steven Murawski, *Stack Exchange*
- **Introduction to Chef** by Nathen Harvey, *Opscode*

The **3-day Technical Program** includes:

- Plenaries by Hilary Mason, *bitly,* and Todd Underwood, *Google*
- Invited Talks by industry leaders such as Ariel Tseitlin, *Netflix;* Jeff Darcy, *Red Hat;* Theo Schlossnagle, *Circonus;* Matt Provost, *Weta Digital;* and Jennifer Davis, *Yahoo!*
- Paper presentations, workshops, vendor exhibition, posters, Guru Is In sessions, BoFs, and more!

New for 2013: The LISA Lab Hack Space!

Register by October 15 and save. Additional discounts are available!

www.usenix.org/lisa2013

Sponsored by

in cooperation with LOPSA

Valerie Aurora on File Systems and the Ada Initiative
An Interview

RIKKI ENDSLEY AND VALERIE AURORA

Rikki Endsley is the Managing Editor of *;login:* and the USENIX Association's Community Manager. rikki@usenix.org

Valerie Aurora (formerly Val Henson) is the Executive Director and co-founder of the Ada Initiative, a nonprofit dedicated to promoting women in open tech/culture. She is an experienced software engineer and was a leading file systems developer, researcher, and consultant for more than a decade. She invented several new file systems concepts, including a widely used power-saving feature in file systems called relative atime, and co-founded the Linux Storage and File Systems Summit.
valerie@adainitiative.org

In July, Valerie Aurora received a 2013 O'Reilly Open Source Award for her long-time contributions to the Linux community and for advocating new developments in Linux file systems. Valerie's expertise in the area of file systems dates back to 2002, when she worked as a software engineer on ZFS at Sun Microsystems. She went on to design and implement chunkfs, union mounts, fsck parallelization, and relative access time (relatime). In January 2011, Valerie announced the launch of the Ada Initiative [1], which helps women get into and stay involved in free and open source projects. Valerie also has a long history with the USENIX Association and our conferences. She has been on the committee for USENIX events, including FAST '09 and the 2007 Linux Storage & Filesystem Workshop, which was co-located with the 5th USENIX Conference on File and Storage Technologies (FAST '07). At HotDep '06, Valerie and her co-authors presented a paper on chunkfs [2].

I've followed Valerie's career and her efforts to help open source communities become more inviting to women since I ran across her "HOWTO Encourage Women in Linux" document during my thesis research in 2006. The document played a huge part in my research into how I, as a tech journalist, could help women in tech by covering their projects and careers. Today the howto is still a thorough, practical resource for encouraging women in IT and reads as if it were written recently. On one hand, "HOWTO Encourage Women in Linux" shows Valerie's forethought on the topic; on the other hand, it shows how much more work needs to be done to make IT more inviting to women.

In this interview, Valerie discusses her work with Linux file systems, conferences, and the Ada Initiative.

Rikki: In 2011, you left your Linux kernel developer position at Red Hat and helped launch the Ada Initiative, but you still do consulting work. What kind of consulting work are you doing?

Valerie: Well, if you look at my consulting Web site, I'm obviously not doing Web design. I do short-term contracts mainly in the area of Linux storage and file systems, usually things like debugging silent data corruption, analyzing performance, and prototyping new features. My favorites are finding the root cause of data corruption and debugging race conditions.

Rikki: Are you doing any Linux/UNIX development right now?

Valerie: Not at the moment. My most recent contracts were all performance tuning or fixing data corruption. My last mainstream kernel patch was to fix a kernel configuration error, and before that a bug fix for a file system freeze locking problem.

Rikki: You were a Linux kernel developer. The kernel team doesn't have a reputation for being particularly inviting. What was your experience like working on the Linux kernel and working with other kernel developers?

Valerie: The vast majority of Linux kernel developers I worked with were incredibly kind, thoughtful, intelligent people who just wanted to make Linux better. Unfortunately, it only takes a few jerks to make a working environment terrible, especially when they are in leadership positions. In my experience, at least 95% of kernel developers I know wish they didn't have to be humiliated and mocked in order to contribute to Linux. But like them, I felt helpless to change a system that condones and rewards nasty behavior, starting from the top. I hope that the work I do in the Ada Initiative will someday help change the Linux kernel development culture, because there were a lot of things I enjoyed about working in the kernel community and I would love to return.

Rikki: Linux Software Engineer at Intel and Linux kernel contributor Sarah Sharp recently called out Linus Torvalds (and others) for…um…less than professional communication on the kernel list. I assume you followed that whole exchange [3]? What are your thoughts on the incident? Do you agree with Sarah's request for everyone to "Keep it professional on the mailing lists"? And what do you think about how Linus handled the situation?

Valerie: I'm one of hundreds of Linux kernel developers, past and present, who agree with Sarah Sharp's request—she's just the person brave enough to directly call for change from Linus Torvalds and other community leadership. I was a little horrified to see how many top-notch kernel developers spoke up to say that this is one reason why they dropped out of kernel development. So I'm thrilled to hear this will be a topic of discussion at the next Linux Kernel Summit. I hope that other kernel developers will join her in standing up for a working environment without abuse.

I think Linus responded based on the information he has. For example, he's probably not aware of research showing that people's intuition that performance improves after severely criticizing someone is wrong: any improvement in performance is due to random chance, what many people are familiar with as "regression to the mean." It turns out that when you evaluate the effect of criticism vs. praise on performance scientifically, praise is the clear winner [5]. We as computer programmers should use the same scientific logical approach to community management as we do for software development.

Rikki: Let's talk file systems. Are you still working on union mounts?

Valerie: No, I already have one full-time job at the Ada Initiative! David Howells is continuing work on union mounts and doing a great job. Sometimes I wonder if the main use of the union mounts project is to find and fix lurking bugs in the VFS code. Certainly working on union mounts is a great way to understand the design and rules of the VFS.

Rikki: Are you working on any other file system-related projects? What's "next" for file systems?

Valerie: At this point, I just applaud from afar whenever Linux file systems hit another milestone. Getting btrfs stable and ready for production is in my mind the top priority for now. I sympathize with their main showstopper bug right now because when I left ZFS development, we were working on the same problem. With a copy-on-write file system, you have to be sure that there is enough space on disk to write out all the changes in the current transaction, before you free the blocks with the original data. It is hard to predict how much space you'll need to do this, so the default is to overestimate the space. (If you underestimate the space, the file system gets wedged and you'll probably have to reboot.) The problem with overestimation, of course, is that writes fail with ENOSPC (out-of-space error) when there is really plenty of disk space left. That's better than crashing but not good. I don't know what the ZFS solution was, but perhaps it could be shared with the btrfs developers.

Rikki: You've been on several USENIX conference committees, including the 2007 Linux Storage & Filesystem Workshop. More recently, you've helped launch the AdaCamps and Allies workshops under the Ada Initiative umbrella. Tell us about those events. What inspired you to start them, and do you think they've been successful? Is an AdaCon in the works?

Valerie: I organized the first Linux File Systems Workshop in 2006 because it was clear that Linux file systems development was stalled, and I wanted it to get moving again. It worked—the formal ext4 development branch was announced two weeks after the 2006 workshop, and the first btrfs announcement came three months after the 2007 summit. Chris Mason explicitly credited that meeting with inspiring btrfs. This meeting continues being useful today, under the ungainly but accurate name of Linux Storage, File Systems, and Memory Management Summit.

AdaCamp is a medium-sized—about 200 people—unconference. My co-founder Mary Gardiner and I started AdaCamp with the goal of increasing women's commitment to open technology and culture. Women are excited by and want to be part of open source or Wikipedia or what have you, but then get more or less subtle "Get out, you don't belong here" messages from their communities. AdaCamp brings women together to support each other in their enthusiasm and commitment to open tech/culture. I thought it would be hard to tell whether AdaCamp worked, but it's been pretty easy. AdaCampers email us regularly to tell us they landed an open source internship, or started women's edit-a-thons in India, or learned how to solder. Our post-AdaCamp surveys show that 85% of attendees thought that AdaCamp increased their commitment to open tech/culture; we're not sure if the other 15% were just already highly committed.

The Allies Workshop was inspired by a workshop taught by Caroline Simard at Grace Hopper Celebration. Her workshop taught men how to support women in tech by role-playing through common scenarios, like having a woman's idea credited to a man in a meeting. We changed it to be short discussions about scenarios in which men could intervene to make women feel more welcome: how to introduce yourself to a woman at a conference, how to respond to harassment on IRC, how to respond to a sexist argument on a Wikipedia talk page. The most interesting success of the Allies Workshop was unexpected: it gave co-workers an opportunity to talk candidly and safely about their experiences with sexism. People who had worked together for years would discover for the first time that their female colleagues were regularly harassed online or propositioned in the lunch room. It gives people a chance to learn and ask questions in a non-judgmental setting.

We're not sure how to do an AdaCon yet but we'd like to have the chance. AdaCon would be a much larger (400?) person conference with more structure and a lower bar to entry. Some of the features of AdaCamp that people love are hard to scale. For example, AdaCamp's open application/invitation-only selection process is an overwhelming amount of work for 200 attendees, but it also creates a safe, welcoming, productive environment that attendees love.

Rikki: The Ada Initiative anti-harassment work seems to be gaining traction, with more than 100 conferences adopting anti-harassment policies in the past few years. Obviously, adopting a policy doesn't "fix" everything. What's next for conferences? What else should conference organizers be doing to make their events inviting and safe for a diverse group of attendees?

Valerie: PyCon US 2013, Open Source Bridge, and AdaCamp are three conferences that show what the next steps are in concrete form. PyCon US did "all the things": anti-harassment policy; outspokenly pro-women community leaders; travel scholarships; inviting women to speak personally; organizing women's events; free booths for women's groups; and explicitly women-friendly spaces, like the Women's Office Hours room and the Ada Initiative Feminist Hacker Lounge. They had 20% women speakers and attendees at a 3000+ person conference.

Open Source Bridge is an explicitly social justice oriented open source conference in Portland, Oregon. The organizers are themselves fairly diverse, and that is reflected in the breadth of topics they cover and the speakers they attract. Like many open source conference organizers, they view their conference as an opportunity to promote social justice and diversity in open source, and spend time and money to accomplish that. This year they improved the accessibility of their conference by adding "travel lanes" with blue tape on the floor, which let people who use mobility devices or have vision impairment move more easily around the conference and around everyone else, too. I hear estimates that attendees are around 30% women and speakers are around 40%.

AdaCamp has a pretty good record of being diverse in a number of dimensions, relative to most open tech/culture events: age, race, place of origin, first language, etc. Being mostly women, we were actually not that diverse in gender per se, though more so in gender identification and expression.

From my experience, the general principles of attracting a diverse audience are:

1. Have a diverse organizing committee.
2. Ask for, listen to, and implement suggestions ASAP.
3. Communicate early and often how much you appreciate diversity of attendees.

For example, a surprisingly effective way to improve diversity is by having a variety of food that caters to people's dietary requirements. At the last AdaCamp, we had people who were vegan, gluten-free, fructose-intolerant, celiac, and allergic to mushrooms, soy, and lettuce, to name a few. We had to fight with the caterers for weeks to get food that had something edible for everyone, tasted great, and was cheap enough for a nonprofit budget. It wasn't easy to accomplish, but that's kind of the point: the fact that we cared enough to go to the effort to make sure everyone could eat lunch together was a signal to our attendees that we cared about their needs. Think about it—if you have to leave the conference to get lunch or snacks, what kind of slap in the face does that feel like? And once you get used to putting in the time and effort to meet people's food needs, it becomes a habit to do so in other ways.

Rikki: Your work with the Ada Initiative has allowed you to meet a variety of interesting people. Has anyone stood out as being particularly effective or innovative when it comes to encouraging women in technology? Which organizations or events do you think are standing out when it comes to encouraging women in tech?

Valerie: Wow, hard question. How do I mention just a few? I apologize to everyone I left out in this answer—I assume *;login;* can't publish a novella!

The Outreach Program for Women [4] (formerly GNOME Outreach Program for Women) has been a stellar success for training and recruiting women developers in open source. OPW is the product of many people's hard work, with Marina Zhurakhinskaya currently leading the project. Increasingly, whenever I meet a new woman open source developer, it turns out she got her start through an OPW internship. Thanks to the Linux Foundation and Sarah Sharp, this year the OPW awarded seven Linux kernel internships [5].

The Geek Feminism Wiki and blog have been around for years, but are starting to reach their full potential as tools to support women in tech. In the past year, the Geek Feminism Wiki has been cited by several mainstream media outlets, such as *The New Yorker*, as supporting evidence for stories on women in tech. In particular, the "Timeline of Incidents" has been crucial supporting evidence in many discussions of whether or not women in tech face systemic discrimination. Geek Feminism is the work of many people, but two of the most prolific contributors and founders are Alex "Skud" Bayley and Mary Gardiner. Without the Geek Feminism Wiki, the Ada Initiative and many other groups could not have made the progress we have over the past few years.

Rikki: What else is the Ada Initiative focusing on right now?

Valerie: We'd like to expand the use of the Allies Workshop as corporate training for technology companies. Many corporations want to hire more women in tech, but aren't aware of the ways their internal culture are off-putting or frankly hostile to women. If companies succeed in hiring women despite their culture, those women often end up fighting an uphill battle to change the internal culture, and often end up leaving out of exhaustion. With the Allies Workshop, we teach men how to fight these battles and change their culture to be more welcoming to women and many different kinds of people. It's fun, too; after one Allies Workshop, an attendee asked HR if they could get "more training like that."

We'd like to contribute to the trend of community-wide codes of conduct going on in open source, Wikipedia, and similar online communities. This is not as easy as banging out an example anti-harassment policy like we did for conferences—at heart, conferences have much more in common with each other than open tech/culture communities. For example, conferences usually have clear-cut leadership who can kick people out of a clearly defined physical space. Online communities are much more varied in governance and structure, so there's no one-size-fits-all way of effectively implementing a code of conduct.

Rikki: Did you know that I referenced your "HOWTO Encourage Women in Linux" [6] article in my Master of Science in Journalism thesis? Even though you wrote the document back in 2002, it passed the test of time and you covered topics that are still relevant today. If you were to update the document now, what (if anything) would you add or change?

Valerie: No, I had no idea—thanks for letting me know! Dozens of people helped me write that HOWTO, mostly other LinuxChix members, and I'm glad so many people found it useful. That HOWTO showed me how powerful the written word could be, and I've never forgotten that lesson.

For years, I had "Update HOWTO Encourage Women in Linux" on my to-do list, but the thought of reading something I'd written that long ago made me cringe. I did finally bring myself to reread it a couple of years ago and was pleasantly surprised with how much of it I still agreed with, enough that I stopped planning to update it. I'd improve some language, I'd replace the example of sexism in the introduction with a link to the Geek Feminism Wiki, and maybe add a few more items. But overall, I think my time is better spent on new projects than in bikeshedding one that is successful enough.

Rikki: Can you tell us about your interest in labyrinths? I know that a lot of developers are also runners. When I interviewed Nick Lang and Jacob Kaplan-Moss about the PyCon 2012 5k [7], Nick told me that running helps him figure out programming problems he's stuck on. Do you feel that way about labyrinths?

Valerie: As a programmer, I'm deeply interested in ways to encourage that unconscious intuitive leap that shows you the bug fix or the solution to the design problem. A common theme in people's stories about "Aha!" moments is that they were doing something else at the time that didn't take up all of their concentration. In my experience, that something else is often walking—but also showering, driving, falling asleep, etc. For me, any kind of walking helps me come up with creative solutions or new insights. Labyrinths are neat both because they let you walk without needing a destination, but also because they have so much history tied up in them.

References

[1] Ada Initiative: http://adainitiative.org/.

[2] Val Henson et al., "Chunkfs: Using Divide-and-Conquer to Improve File System Reliability and Repair": https://www.usenix.org/conference/hotdep-06/chunkfs-using-divide-and-conquer-improve-file-system-reliability-and-repair.

[3] Linux-kernel list email exchange: http://marc.info/?l=linux-kernel&m=137390362508794&w=2.

[4] Outreach Program for Women: https://wiki.gnome.org/OutreachProgramForWomen.

[5] Linux Kernel Internships 2013: http://sarah.thesharps.us/2013/05/23/%EF%BB%BF%EF%BB%BFopw-update/.

[6] Val Henson, "HOWTO Encourage Women in Linux": http://tldp.org/HOWTO/Encourage-Women-Linux-HOWTO/.

[7] Rikki Endsley, "How to Pound the Pavement with Programmers at PyCon" (PyCon 2012 5K), *Network World*, Mar. 6, 2012: http://www.networkworld.com/community/blogs/pycon5k.

Modular SDN Programming with Pyretic

JOSHUA REICH, CHRISTOPHER MONSANTO, NATE FOSTER,
JENNIFER REXFORD, AND DAVID WALKER

Joshua Reich is an NSF/CRA Computing Innovation Fellow at Princeton University's Department of Computer Science. He designs and builds systems to utilize networks more effectively—currently focusing on SDNs. His work on Pyretic received the NSDI Community Award (shared with this article's co-authors).
jreich@cs.princeton.edu

Christopher Monsanto is a Ph.D. candidate at Princeton University, advised by David Walker. His research interests include programming languages and distributed computing. chris@monsan.to

Nate Foster is an Assistant Professor of Computer Science at Cornell University. His research focuses on abstractions and tools for building reliable systems.
jnfoster@cs.cornell.edu

Jennifer Rexford is the Gordon Y.S. Wu Professor of Engineering in the Computer Science Department at Princeton University. She previously worked at AT&T Research, where she designed network-management techniques that were deployed in AT&T's backbone network. jrex@cs.princeton.edu

David Walker is a Professor of Computer Science at Princeton University. His research focuses on the theory, design, and implementation of programming languages. dpw@cs.princeton.edu

Software-Defined Networking (SDN) enables innovation in network management by giving a programmable controller direct control over the underlying switches through an open, standard API, like OpenFlow. However, existing SDN controllers offer programmers a low-level programming interface akin to assembly language. In this article, we present Pyretic, a programming platform that raises the level of abstraction and enables the creation of modular software, allowing programmers to create sophisticated SDN applications.

Managing today's computer networks is a complex and error-prone task. These networks consist of a wide variety of devices, from routers and switches to firewalls, network-address translators, load balancers, and intrusion-detection systems. Network administrators must express policies through tedious box-by-box configuration, while grappling with a multitude of protocols and baroque, vendor-specific interfaces.

In contrast, Software-Defined Networking (SDN) is redefining the way we manage networks. In SDN, a controller application uses a standard, open interface, such as OpenFlow [1], to specify how network elements or switches should handle incoming packets. Programmers develop their own new controller applications on top of a controller platform, which provides a programming API built on top of OpenFlow. Separating the controller platform and applications from the network elements allows anyone—not just the equipment vendors—to program new network control software.

In just a few years, SDN has enabled a wealth of innovation, including prominent commercial successes such as Nicira's network virtualization platform and Google's wide-area traffic-engineering system. Most of the major switch vendors support the OpenFlow API, and many large information-technology companies are involved in SDN consortia, such as the Open Networking Foundation and the Open Daylight initiative.

SDN is creating exciting new opportunities for network-savvy software developers and software-savvy network practitioners alike. But how should programmers write these controller applications? The first generation of SDN controller platforms offer programmers a low-level API closely resembling the interface to the switches. This forces programmers to program in "assembly language," by manipulating bit patterns in packets and carefully managing the shared rule-table space.

In the Frenetic Project [2], we are designing simple, reusable, high level abstractions for programming SDNs; and efficient runtime systems that automatically generate and install the corresponding low-level rules on switches [3–7]. Our abstractions cover the main facets of managing a network-specifying packet-forwarding policy, monitoring network conditions, and dynamically updating policy to respond to network events. In this article, we describe Pyretic, our Python-based platform that embodies many of these concepts, and enables systems programmers to create sophisticated SDN applications.

Pyretic is open-source software that offers a BSD-style license compatible with the needs of both commercial and research developers. Both the source code for, and a pre-packaged VM

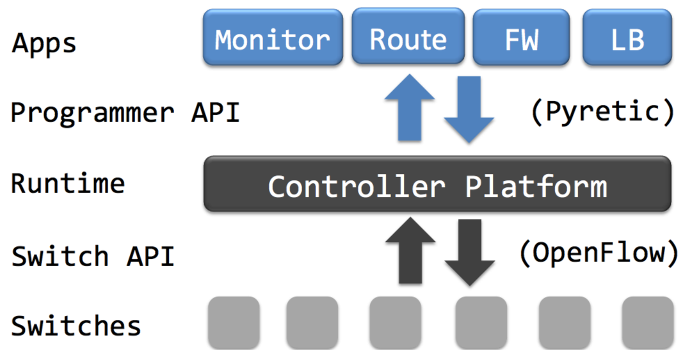


Figure 1: Software Defined Network (SDN)

containing, Pyretic's core policy language, libraries, and runtime are available on the Pyretic home page [8], along with documentation, video tutorials, links to our email discussion list, and more. Feel free to download and run any of the Pyretic examples covered in the article.

OpenFlow

Pyretic is both a response to the shortcomings of OpenFlow as a programmer API, and a client of OpenFlow in its role as an API to network switches. As such, we begin with a brief review of OpenFlow.

OpenFlow Switches

An OpenFlow switch has a rule table, where each rule includes:

- a bit pattern: including wildcards, for matching header fields—for example, MAC and IP addresses, protocol, TCP/UDP port numbers, physical input port, etc.;
- a priority: to break ties between overlapping patterns;
- a list of actions: for example, forward out a port, flood, drop, send to controller, assign a new value to a header field, etc.;
- optional hard and soft timeouts to evict stale rules;
- byte and packet counters that collect information about how much traffic is flowing through each rule.

Upon receiving a packet, the switch finds the highest-priority matching rule, applies each action, and updates the counters. Newer versions of OpenFlow support additional header fields and multiple stages of tables.

OpenFlow Controllers

The OpenFlow protocol defines how the controller and switches interact. The controller maintains a connection to each switch over which OpenFlow messages are sent. The controller uses these OpenFlow messages to (un)install rules, query the traffic counters, learn the network topology, and receive packets when the switch applies the "send to controller" action. Most existing controller platforms offer programmers an API that is a thin "wrapper" around these operations. Applications are expressed as event handlers that respond to events such as packet arrivals, topology changes, and new traffic statistics.

Controller Applications

OpenFlow has enabled a wealth of controller applications, including flexible access control, Web server load balancing, energy-efficient networking, billing, intrusion detection, seamless mobility and virtual-machine migration, and network virtualization. As an example, consider "MAC learning"—an application designed to detect the arrival of new hosts, discover their MAC addresses, and route packets to them. To begin, the application starts by installing a default rule in each edge switch that matches all packets and sends them to the controller. Upon receiving a packet, the application learns the location (i.e., the switch and input port) of the sender. If the receiver's location is already known, the application installs rules that direct traffic in both directions over a shortest path from one to the other; otherwise, the application instructs the switch to flood—broadcasting the packet to all possible receivers. If a host moves to a new location, the default rule at the new switch sends the next packet to the controller, allowing the application to learn the host's new location and update the paths that carry traffic to and from the host. Consequently, hosts can continue communicating without disruption, even when one or both hosts move.

Pyretic Language

Pyretic encourages programmers to focus on how to specify a network policy at a high level of abstraction, rather than how to implement it using low-level OpenFlow mechanisms. In particular, instead of implementing a policy by incrementally installing physical rule after physical rule on switch after switch, a Pyretic policy is specified for the entire network at once, via a function from an input located packet (i.e., a packet and its location) to an output set of located packets. The output packets can have modified fields and usually end up at new locations—this is how packet forwarding occurs. The programmer does not need to worry about which OpenFlow rules are used to move packets from place to place.

One of the primary advantages of Pyretic's policies-as-abstract-functions approach to SDN programming is that it helps support modular programming. In traditional OpenFlow programming, the programmer cannot write application modules independently

without worrying that they might interfere with one another. Rather than forcing programmers to carefully merge multiple pieces of application logic by hand, a Pyretic program can combine multiple policies together using one of several *policy composition operators*, including *parallel composition* and *sequential composition*.

On existing SDN controller platforms, monitoring is merely a side-effect of installing rules that send packets to the controller, or accumulate byte and packet counters. Programmers must painstakingly create rules that simultaneously monitor network conditions and perform the right forwarding actions. Instead, Pyretic integrates monitoring into the policy function and supports a high level query API. The programmer can easily combine monitoring and forwarding using parallel composition. Pyretic also provides facilities for creating a dynamic policy whose behavior will change over time, as specified by the programmer. Composition operators can be applied to these dynamic policies just as easily as fixed static ones.

Finally, Pyretic offers a rich topology-abstraction facility that allow programmers to apply policy functions to an abstract view of the underlying network. This facility is particularly noteworthy in that it is actually an application built on top of Pyretic using the other abstractions in the language.

In this section, we illustrate the features of the language using examples. Along the way, we build toward a single-switch Pyretic application that dynamically splits incoming traffic across several server instances. We conclude by using topology abstraction to distribute this single-switch application across a network of many switches.

Network Policy as a Function

A controller application determines the policy for the network at any moment in time. A conventional OpenFlow program includes explicit logic that creates and sends rule-installation messages to switches (logic that includes defining the low-level bit-match patterns, priorities, and actions for these rules) and that registers callbacks that poll traffic counters and handle packets sent to the controller.

In contrast, Pyretic hides these low-level details by allowing programmers to express policies as compact, abstract functions that take a packet (at a given location) as input, and return a set of new packets (at potentially different locations). Returning the empty set corresponds to dropping the packet. Returning a single packet corresponds to forwarding the packet to a new location. Returning multiple packets corresponds to multicast.

The simplest possible Pyretic policy is one where every switch floods each packet out all ports on the network spanning tree. In conventional OpenFlow programming, the controller application would, for each switch, install the rule whose pattern is “don't care” on all bits, with a single action “flood” (if that action is even supported by the switch). In contrast, in Pyretic, the programmer simply writes one line:

```
flood()
```

where flood() is interpreted as a function that takes a packet located at any port on any switch in the network as an input and outputs zero, one, or more copies of the same packet at the output ports of the switch it arrived at—one packet for each port on the network's spanning tree. Hence, this simple policy will allow any collection of hosts to broadcast information to one another over a network. Moreover, the policy no longer depends upon specific switch features. The switches used need not implement a “flood” primitive themselves as the runtime system can choose to implement flooding behavior using other OpenFlow actions—a good thing because the “flood” action is an optional feature in OpenFlow 1.0.

Of course, Pyretic programmers will typically write much more sophisticated policies. Here's a fragment of a policy that uses several more Pyretic features to route a packet with destination IP 10.0.0.1 across switches A and B.

```
(match(switch=A) & match(dstip='10.0.0.1') >> fwd(6)) +
(match(switch=B) & match(dstip='10.0.0.1') >> fwd(7))
```

Here, we use *predicate policies* (including match and conjunction) to disambiguate between packets based on their location in the network as well as their contents; we use *modification policies* (such as `fwd`) to change the header content or location of packets; and we use *composition operators* (such as +, parallel composition and >>, sequential composition) to put together policy components. Each of these features, as well as others, will be explained in the upcoming sections; Table 1 lists several of the most common basic Pyretic policies.

In this slightly more elaborate policy, there are components that look somewhat like OpenFlow rules—they match different kinds of packets and perform different actions; however, as the simpler flood example shows, these policies do not necessarily map to OpenFlow rules in a one-to-one fashion. Consequently, Pyretic programmers must discard the rule-based mental programming model and adopt the functional one. We believe doing so encourages programmers to focus their minds entirely on the essential problem: determining the fundamental, high-level logic required to implement the application properly, not the low-level encoding of that logic in terms of hardware abstractions and a series of controller-level event handlers. This also leads to much more concise code, avoids replicating related functionality, and reduces the risk of accidental inconsistencies between different parts of the application.

Syntax	Summary
`identity`	returns original packet
`drop`	returns empty set
`match(f=v)`	`identity` if field `f` matches `v`, `drop` otherwise
`modify(f=v)`	returns packet with field `f` set to `v`
`fwd(a)`	`modify (port=a)`
`flood()`	returns one packet for each local port on the network spanning tree

Table 1: Selected policies

From Bit Patterns to Boolean Predicates

An OpenFlow rule matches packets based on a bit pattern in the header fields, where each bit is a 0, 1, or "don't care"; however, expressing a policy in terms of bit patterns is tedious. For example, matching all packets except those with a destination IP address of 10.0.0.1 requires two rules. The first, higher-priority rule matches all packets destined to 10.0.0.1, so that all remaining packets "fall through" to the second, lower-priority rule that has a wildcard in each bit position. Similarly, matching either 10.0.0.3 or 10.0.0.4 requires two rules, one for each IP address (as there is no single bit-pattern that matches both).

Instead of bit patterns in packet-header fields, Pyretic allows programmers to write basic predicates of the form `match(f=v)`, demanding that a field `f` match an abstract value `v` (such as an IP address). They can then construct more complicated predicates using standard Boolean operators such as and (&), or (|), and not (~). Intuitively, all these predicates act as filters: If the incoming packet satisfies the predicate, the packet passes through the filter untouched, presumably to be processed in some interesting way by some subsequent part of the policy. If the incoming packet does not satisfy the predicate, it is dropped (i.e., the empty set of packets is generated as a result). For example, the Pyretic programmer simply writes

```
~match(dstip='10.0.0.1')
```

or

```
match(switch=A) &
 (match(dstip='10.0.0.3') | match(dstip='10.0.0.4'))
```

and the runtime system ensures that packets are filtered accordingly.

Virtual Packet Header Fields

A policy function in Pyretic can match on a packet-header field (using `match(f=v)`), and can assign a new value to a header field (using `modify(f=v)`). As we have seen, the fields available to the programmer include the standard physical OpenFlow packet header fields, such as source and destination IP; however, unlike OpenFlow packets, Pyretic packets provide a single unified abstraction for both the packet and its associated metadata. To this end, Pyretic packets also include standard virtual fields switch and port that together specify a packet's location in the network. In fact, the `fwd` policy we saw previously is actually just a special case of `modify`! Reassigning the value of port simply "moves" the packet from the port on which it arrived to the port on which it will be sent. The burden of managing all the details needed to ensure that each packet is forwarded out the correct hardware port is left to the Pyretic runtime.

Finally, Pyretic programmers are free to define their own, new virtual fields and use them however they choose, treating each Pyretic packet as if it were a Python dictionary. For example, a programmer may want to assign a packet to one of several paths through a network. Tagging the packet with the chosen path makes it easier to direct the packet over each of the hops in the path. In Pyretic, the programmer could create a new path field and assign it a particular path identifier. Here again, the burden of realizing this falls to the Pyretic runtime, which might, under the hood, represent the appropriate information using a conventional packet tagging mechanism such as VLANs or MPLS labels.

Parallel and Sequential Composition

A controller application often needs to perform multiple tasks (e.g., routing, server load balancing, monitoring, and access control) that affect handling of the same traffic. Rather than writing one monolithic program, programmers should be able to combine multiple independently written modules together. In traditional OpenFlow programming, different modules could easily interfere with each other. One module might overwrite the rules installed by another, or drop packets another module expects to see at the controller. Instead, Pyretic offers two simple composition operators that allow programmers to combine policies in series or in parallel.

SEQUENTIAL COMPOSITION

Sequential composition (>>) treats the output of one policy as the input to another. Consider a simple routing policy:

```
match(dstip='2.2.2.8') >> fwd(1)
```

In this policy, the match predicate filters out all packets that do not have destination 2.2.2.8. The >> operator places this filter in sequence with the forwarding policy `fwd(1)`. Hence any packets that pass through the filter are forwarded out port 1. Likewise, the programmer may write

```
match(switch=1) >> match(dstip='2.2.2.8') >> fwd(1)
```

to specify that packets located at switch 1 and destined to IP address 2.2.2.8 should be forwarded out port 1. This code uses

sequential composition to compose three independent policies. The first two policies happen to be filters (though they may be arbitrary policies). Of course, filtering packets first by one condition and then by a second condition is equivalent to filtering packets by the conjunction (&) of the two conditions.

PARALLEL COMPOSITION

Parallel composition (+) applies two policy functions on the same packet and combines the results. For example, a routing policy `R` could be expressed as

```
R = (match(dstip='2.2.2.8') >> fwd(1)) +
    (match(dstip='2.2.2.9') >> fwd(2))
```

Those packets destined to 2.2.2.8 will be forwarded out port 1, while those destined to 2.2.2.9 will be forwarded out port 2.

As another example, consider a server load-balancing policy that splits request traffic directed to destination 1.2.3.4 over two backend servers (2.2.2.8 and 2.2.2.9), depending on the first bit of the source IP address (packets with sources starting with 0 fall under IP prefix 0.0.0.0/1 and are routed to 2.2.2.8). This results in the policy:

```
L = match(dstip='1.2.3.4') >>
     ((match(srcip='0.0.0.0/1') >> modify(dstip='2.2.2.8')) +
      (~match(srcip='0.0.0.0/1') >> modify(dstip='2.2.2.9')))
```

This policy happens to adhere to a particularly common pattern: a clause matching one predicate is immediately followed by a clause matching its negation. Of course, in conventional programming languages, such patterns are just if statements. In Pyretic, `if_` is an abbreviation that makes policies easier to read:

```
L = match(dstip='1.2.3.4') >>
    if_(match(srcip='0.0.0.0/1'),
        modify(dstip='2.2.2.8'),
        modify(dstip='2.2.2.9'))
```

CODE REUSE

One final example highlights the power of Pyretic's composition operators to enable modular programming. In just one line, the programmer can write

```
L >> R
```

producing a new policy that first selects a server replica and then forwards the traffic to that chosen replica. As simple as it seems, this kind of composition is impossible to achieve when programming directly against the OpenFlow API.

Traffic Monitoring

In traditional OpenFlow programs, collecting traffic statistics involves installing rules (so that byte and packet counters are available), issuing queries to poll these counters, parsing the responses when they arrive, and combining counter values across multiple rules.

Syntax	Summary
`packets( limit=n, group_by=[f1,f2,...])`	callback on every packet received for up to `n` packets identical on fields `f1,f2,...`
`count_packets( interval=t, group_by= [f1,f2,...])`	count every packet received callback every `t` seconds providing count for each group
`count_bytes( interval=t, group_by=[ f1,f2,...])`	count every byte received callback every `t` seconds providing count for each group

Table 2: Query policies

In Pyretic, network monitors are just another simple type of policy that may be conjoined to any of the other policies seen so far. Table 2 shows several different kinds of monitoring policies available in Pyretic, including policies that monitor raw packets, packet counts, and byte counts. The forwarding behavior of these policies is the same as a policy that drops all packets.

For example, a programmer may create a new query for the first packet arriving from each unique source IP

```
Q = packets(limit=1,group_by=['srcip'])
```

and restrict it to Web-traffic requests (i.e., packets destined to TCP port 80):

```
match(dstport=80) >> Q
```

To print each packet that arrives at Q, the programmer registers a callback routine to handle Q's callback,

```
def printer(pkt):
  print pkt

Q.register_callback(printer)
```

The runtime system handles all of the low-level details of supporting queries—installing rules, polling the counters, receiving the responses, combining the results as needed, and composing query implementation with the implementation of other policies. For example, suppose the programmer composes the example monitoring query with a routing policy that forwards packets based on the destination IP address. The runtime system ensures that the first TCP port 80 packet from each source IP address reaches the application's printer routine, while guaranteeing that this packet (and all subsequent packets from this source) is forwarded to the output port indicated by the routing policy.

Writing Dynamic Policies

Query policies are often used to drive changes to other dynamic policies. These dynamic policies have behavior (defined by `self.policy`) that changes over time, according to the programmer's specification.

For example, the routine `round_robin` takes the first packet from a new client (source IP address) and updates the policy's behavior (by assigning `self.policy` to a new value), so all future packets from this source are assigned to the next server in the sequence (by rewriting the destination IP address); packets from all other clients are treated as before. After updating the policy, `round_robin` also moves the "currently up" server to the next server in the list.

```
def round_robin(self,pkt):
  self.policy = if_(match(srcip=pkt['srcip']),
                    modify(dstip=self.server),
                    self.policy)
  self.client += 1
  self.server = self.servers[self.client % m]
```

The programmer creates a new "round-robin load balancer'' dynamic policy class `rrlb` by subclassing `DynamicPolicy` and providing an initialization method that registers `round_robin` as a callback routine:

```
class rrlb(DynamicPolicy):
  def __init__(self,s,servers):
    self.switch = s
    self.servers = servers
    ...
    Q.register_callback(self.round_robin)
    self.policy = match(dstport=80) >> Q

  def round_robin(self,pkt):
     ...
```

Note that here the query Q is defined as in the previous subsection; the only difference is that the programmer registers `round_robin` as the callback, instead of printer. The programmer then creates a new instance of `rrlb` (say, one running on switch 3 and sending requests to server replicas at 2.2.2.8 and 2.2.2.9) in the standard way

```
servers = ['2.2.2.8','2.2.2.9']
rrlb_on_switch3 = rrlb(3,servers)
```

producing a policy that can be used in exactly the same ways as any other. For example, to compose server load balancing with routing, we might write the following:

```
rrlb_on_switch3 >> route
```

Topology Abstraction

In traditional OpenFlow programming, a controller application written for one switch cannot easily be ported to run over a distributed collection of switches, or be made to share switch hardware with other packet-processing applications. In the case of our load balancer example, we may well want to use it to balance load coming in from many different hosts connected to many different switches in a complex network. And yet, we would prefer to avoid conflating the relatively simple functionality of the load balancer with the logic needed to route the traffic across the network. A good solution to this problem is to use topology abstraction to partition the application into two pieces: one that does the load balancing as before, as if the balancer was implemented on one big switch that could connect all hosts together, and one that decides on the lower level routes that implement it. This also serves a secondary purpose: the load balancer is reusable and can operate over any network of switches.

To develop this kind of modular program, Pyretic offers a library for topology abstraction that can represent multiple underlying switches as a single derived virtual switch, or, alternatively, one underlying switch as multiple derived virtual switches.

For example, to produce a policy that applies the client policy `rrlb_on_switch3` to a derived (i.e., virtual) switch 3 that abstracts switches 1, 2, and 3 as a single merged switch, the programmer simply uses Pyretic's virtualize function, inputting the desired policy function and the topology transformation:

```
virtualize(rrlb_on_switch3,
           merge(name=3,
                 from_switches=[1,2,3]))
```

Here, the `merge` topology transformation takes the name of a single virtual switch and a list of underlying switches that used to create it. Inside, the `merge` transformation applies shortest-path routing to direct packets from one edge link to another over the underlying switches. `merge` encodes this transformation in three auxiliary policies—one that handles incoming traffic, one that handles traffic passing through the derived switch, and one that handles traffic leaving the switch.

The virtualize policy then implements a transformation of the written policies (the client policy and three auxiliary policies) using virtual header fields and sequential composition to produce a single new policy written for the underlying network [6]. The resulting policy is exactly the same as any other Pyretic policy, and can be both composed with other policies, or used as the basis for yet another layer of virtualization.

Pyretic Runtime

Of course, high level programming abstractions are only useful if they can be implemented efficiently on the switches. This section provides a brief overview of the Pyretic runtime system, focusing on the backend interface to the OpenFlow switches and policy evaluation.

Backend Interface

Pyretic's runtime is designed to be used atop a variety of different OpenFlow controller backends. The Pyretic runtime connects via a standard socket to a simple OpenFlow client that could be written on top of any OpenFlow controller platform. The runtime manipulates the network by sending messages to the client (e.g., to inject packets, modify rules, and issue counter reads). Likewise messages from the client keep Pyretic updated regarding network events (e.g., packet ins, port status events, counter values read). This design enables Pyretic to take advantage of the best controller technology available, and allows the system to be entirely self-contained. The current Pyretic runtime comes packaged with an OpenFlow client written on the popular POX controller platform.

Policy Evaluation

The Pyretic runtime implements an interpreter that evaluates an input packet against the current policy. In its simplest mode of operation, all packets are initially evaluated by this interpreter. Concurrently, the runtime keeps track of currently active queries, updates to dynamic policies, and modifications to the network topology. On its general setting, when it is safe to do so, the runtime proactively installs rules on switches before they are needed, to avoid unnecessary switch-controller latency. For more information on the current runtime implementation, please see the Pyretic home page [8].

Conclusions

Pyretic lowers the barrier to creating sophisticated SDN applications and comes with several example of common enterprise and datacenter network applications (e.g., hub, MAC-learning switch, traffic monitor, firewall, ARP server, network virtualization, and gateway router). Since the initial release of Pyretic in April 2013, the community of developers has grown quickly. Some have built new applications from scratch, while others have ported systems originally written on other platforms.

In one case, the Resonance [9] system for event-driven control was rewritten in Pyretic, taking approximately one programmer-day and resulting in a six-fold reduction in code size over an earlier version written on the NOX controller platform. These savings were realized thanks to Pyretic's declarative design and powerful yet concise policy language. Short expressions involving basic policies, such as `match` and `fwd`, combined with composition operators to replace complex code specifying various packet handlers and the logic they contained: packet matching, modification and injection, as well as OpenFlow rule construction and installation. In fact, Pyretic's focus on modular design enabled the Resonance team to encode more sophisticated policies than had been available in the NOX version.

Pyretic has also been featured in Georgia Tech's SDN Coursera course [10] where it was used as the platform for one of the course's three programming assignments.

In addition to enhancing our runtime system with enhanced compilation support, in our ongoing work we are also making extensions to the language and runtime system to support new features, such as quality-of-service mechanisms and parsing of packet contents. Additionally, we are creating more sophisticated applications, including RADIUS and DHCP services (to authenticate end hosts and assign them IP addresses) and wide-area traffic-management solutions for Internet Service Providers at SDN-enabled Internet Exchange Points.

We welcome newcomers to our community, whether they are interested in using Pyretic or in contributing to its development. Please visit our Web site, join our discuss list, or email us.

Acknowledgments

Our work is supported in part by ONR grant N00014-09-1-0770 and NSF grants 1111698, 1111520, 1016937, 1253165, and 0964409, a Sloan Research Fellowship, and a NSF/CRA Computing Innovation Fellowship. Any opinions, findings, and recommendations are those of the authors and do not necessarily reflect the views of the NSF, CRA, ONR, or the Sloan Foundation.

References

[1] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: Enabling Innovation in Campus Networks," SIGCOMM CCR, vol. 38, no. 2 (2008), pp. 69-74.

[2] The Frenetic Project: http://www.frenetic-lang.org.

[3] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker, "Frenetic: A Network Programming Language," ACM ICFP, Sept. 2011.

[4] C. Monsanto, N. Foster, R. Harrison, and D. Walker, "A Compiler and Run-Time System for Network Programs," POPL, Jan. 2012.

[5] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker, "Abstractions for Network Update," ACM SIGCOMM, Aug. 2012.

[6] C. Monsanto, J. Reich, N. Foster, J. Rexford, and D. Walker, "Composing Software-Defined Networks," USENIX NSDI, 2013.

[7] N. Foster, M. J. Freedman, A. Guha, R. Harrison, N. P. Katta, C. Monsanto, J. Reich, M. Reitblatt, J. Rexford, C. Schlesinger, A. Story, and D. Walker, "Languages for Software-Defined Networks," IEEE Communications, vol. 51 (Feb 2013), pp. 128-134.

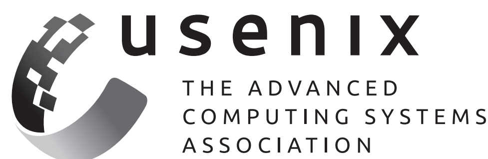
[8] Pyretic home page: http://www.frenetic-lang.org/pyretic.

[9] Resonance Project: http://resonance.noise.gatech.edu.

[10] Coursera course on SDN: https://www.coursera.org/course/sdn.

Do you know about the USENIX Open Access Policy?

USENIX is the first computing association to offer free and open access to all of our conferences proceedings and videos. We stand by our mission to foster excellence and innovation while supporting research with a practical bias. Your membership fees play a major role in making this endeavor successful.



Please help us support open access.
Renew your USENIX membership
and ask your colleagues to join or renew today!

www.usenix.org/membership

Drilling Network Stacks with packetdrill

NEAL CARDWELL AND BARATH RAGHAVAN

Neal Cardwell received an M.S. in Computer Science from the University of Washington, with research focused on TCP and Web performance. He joined Google in 2002. Since then he has worked on networking software for google.com, the Googlebot web crawler, the network stack in the Linux kernel, and TCP performance and testing. neal@google.com

Barath Raghavan received a Ph.D. in Computer Science from UC San Diego and a B.S. from UC Berkeley. He joined Google in 2012 and was previously a Senior Researcher at ICSI in Berkeley, CA. His work has focused on network protocol design, applied cryptography, and sustainable computing. barath@google.com

Testing and troubleshooting network protocols and stacks can be painstaking. To ease this process, our team built packetdrill, a tool that lets you write precise scripts to test entire network stacks, from the system call layer down to the NIC hardware. packetdrill scripts use a familiar syntax and run in seconds, making them easy to use during development, debugging, and regression testing, and for learning and investigation.

Have you ever had the experience of staring at a long network trace, trying to figure out what on earth went wrong? When a network protocol is not working right, how might you find the problem and fix it? Although tools like tcpdump allow us to peek under the hood, and tools like netperf help measure networks end-to-end, reproducing behavior is still hard, and knowing when an issue has been fixed is even harder.

These are the exact problems that our team used to encounter on a regular basis during kernel network stack development. Here we describe packetdrill, which we built to enable scriptable network stack testing. packetdrill allows a user to specify a sequence of interactions with the network stack in a short script and then execute the script to verify the network stack's behavior.

packetdrill has a range of applications that we have been using it for on a daily basis:

- Regression testing a network stack: we have a suite of hundreds of packetdrill scripts that are run by all developers on our team before submitting a patch for review.
- Test-driven development of network protocols: we have developed several new features for Linux TCP using packetdrill.
- Reproduction of bugs seen in production network traces: we have used packetdrill to isolate hard-to-reproduce bugs seen in complex real traces.

We also believe that packetdrill can have significant value for

- self-directed learning of a network protocol, by writing scripts to elicit various behaviors from the network protocol in question;
- as a tool for teaching about network protocols in a university setting; and
- with minor extensions, scriptable testing of network applications that live above core network protocols.

packetdrill currently enables the user to test the correctness, performance, security, and general behavior of core network protocols—TCP, UDP, and ICMP—running on IPv4 and IPv6, and runs on Linux, FreeBSD, NetBSD, and OpenBSD. The tool is primarily for black-box testing, though it provides some support for examining internal network protocol state when supported by the OS.

packetdrill is released under version 2 of the GNU Public License (just like the Linux kernel), and we encourage patches, which you can send to the packetdrill email list (packetdrill@googlegroups.com), to extend the tool. For example, adding support for other IP-based protocols, such as DCCP or SCTP, would be straightforward, and we welcome patches to support these and other protocols.

The packetdrill Scripting Language

The packetdrill scripting language provides all the basic building blocks needed to set up a detailed, reproducible scenario for black-box testing of a network stack. The tool supports four types of statements: packets, system calls, shell commands, and Python scripts. Each statement is timestamped and is executed by the interpreter in real time, verifying that events proceed as the script expects. We discuss each type of statement in turn.

Packets

Arguably the most essential building block of any networking scenario is the packet. packetdrill allows the user to specify both inbound packets to inject into the system under test and outbound packets to expect the system to send. To keep the tests succinct and easy to both write and read, we use a syntax like that of tcpdump, which is familiar to most developers and system administrators who troubleshoot networking issues on UNIX systems. Modeled after UNIX shell input/output redirection operators, < denotes an input packet to construct and inject and > denotes an output packet to sniff and verify.

Here's an example of a TCP SYN packet, which packetdrill creates and injects into the network stack under test 100 ms after the start of the test:

```
0.100 < S 0:0(0) win 32792 <mss 1000,nop,nop,sackOK,nop,wscale 6>
```

Here's an example of an outbound UDP packet expected to be sent immediately after a prior event (denoted by +0), which packetdrill sniffs for and then verifies for matching specification (e.g., length, headers, etc.):

```
+0 > udp (1472)
```

System Calls

System calls are the other essential building block of a black-box network stack test scenario, since they express the application's intent and the work the kernel is supposed to perform. To specify a system call in packetdrill, the user only needs to provide the call's salient inputs, the duration for which the call is expected to block (if at all), and the expected outputs. The syntax mirrors that of strace, which we chose because it is familiar to most Linux users and is clear to any C programmer. In addition, in most cases it provides a quick one-line summary of both the inputs and outputs of a system call.

Here's an example of a bind() system call invocation in packetdrill notation:

```
+0 bind(3, ..., ...) = 0
```

In this example, 3 denotes the file descriptor number to pass in, and the = 0 denotes the expected return value (i.e., the user expects the system call to succeed). The ellipsis (...) here in place of the traditional addr and addrlen parameters is not to simplify the presentation in this article; rather, packetdrill supports this notation, again borrowed from strace, to allow scripts to omit irrelevant details. Under the hood, packetdrill fills in a sockaddr for bind and connect using an IP address and port number from command line options (with defaults for those options chosen to be appropriate for the address family involved—e.g., RFC 1918 private IPv4 address spaces). Hiding these details simplifies scripts and makes them quicker and easier to write and read. Just as important, it allows most scripts to be run without modification using IPv4, IPv6, or dual-mode (AF_INET6 socket with IPv4 traffic), depending on the command line arguments to packetdrill.

Shell Commands

packetdrill also allows scripts to specify arbitrary shell command sequences to execute, typically to configure the machine under test (e.g., with sysctl) or to assess the state of the machine (e.g., with netstat or ss). packetdrill implements this, as you would imagine, using a simple invocation of the C library's system() call. To enclose the commands, packetdrill borrows the backtick syntax used in shells and Perl.

Here's a typical example, which disables TCP timestamps in order to test TCP behavior without them:

```
+0 `sysctl -q net.ipv4.tcp_timestamps=0`
```

Python Commands

Finally, packetdrill allows inline Python code snippets to print information and to make assertions about the internal state of a TCP socket using the TCP_INFO getsockopt() option supported by Linux and FreeBSD. Users can enclose such snippets between `%{` and `}%` tokens, a nod to lex/flex and yacc/bison syntax for embedding inline C snippets.

The following Linux-based example asserts that the sender's congestion window is 10 packets:

```
+0 %{ assert tcpi_snd_cwnd == 10 }%
```

In this example, under the hood packetdrill will make a TCP_INFO getsockopt() call for the socket under test and then stash the output tcp_info struct in memory. Then, when the test finishes execution, packetdrill emits a Python script encoding the contents of the tcp_info struct, followed by the Python code snippet that can print or make assertions about any interesting values.

An Example packetdrill Script

Next we give a short example. Suppose that you want to verify that your TCP stack correctly validates incoming TCP RST packets (see RFC 5961, Improving TCP's Robustness to Blind In-Window Attacks). Listing 1 shows a script (targeted at Linux)

that verifies that a TCP endpoint ignores a RST whose sequence number is just beyond the offered window.

```
// Create a listening TCP socket.
0  socket(..., SOCK_STREAM, IPPROTO_TCP) = 3
+0  setsockopt(3, SOL_SOCKET, SO_REUSEADDR, [1], 4) = 0
+0  bind(3, ..., ...) = 0
+0  listen(3, 1) = 0

// Establish a new connection.
+0  < S 0:0(0) win 32792 <mss 1000,sackOK,nop,nop,nop,wscale 7>
+0  > S. 0:0(0) ack 1 win 29200 <mss
      1460,nop,nop,sackOK,nop,wscale 6>
+.1 < . 1:1(0) ack 1 win 257
+0  accept(3, ..., ...) = 4

// sequence number out of window!
+.010 < R. 29202:29202(0) ack 1 win 257
// verify that the connection is OK
+.010  write(4, ..., 1000) = 1000
+0  > P. 1:1001(1000) ack 1
```

Listing 1: Validating handling of out-of-window RSTs

packetdrill's Design

Execution Model

packetdrill parses an entire test script, and then executes each timestamped line in real time—at the pace described by the timestamps—to replay and verify the scenario. The packetdrill interpreter has one thread for the main flow of events and another for executing any system calls that the script expects to block (e.g., poll()).

For convenience, scripts use an abstracted notation for packets. Internally, packetdrill models aspects of TCP and UDP behavior; to do this, packetdrill maintains mappings to translate between the values in the script and those in the live packet. The translation includes IP, UDP, and TCP header fields, including TCP options such as SACK and timestamps. Thus we track each socket and its IP addresses, port numbers, TCP sequence numbers, and TCP timestamps.

Local and Remote Testing

packetdrill enables two modes of testing: local mode, using a TUN virtual network device, or remote mode, using a physical NIC.

In local mode, packetdrill uses a single machine and a TUN virtual network device as a source and sink for packets. This tests the system call, sockets, TCP, and IP layers, and is easier to use because there is less timing variation, and users need not coordinate access to multiple machines.

In remote mode, users run two packetdrill processes, one of which is on a remote machine and speaks to the system under test over a LAN. This approach tests the full networking system: system calls, sockets, TCP, IP, software and hardware offload mechanisms, the NIC driver, NIC hardware, wire, and switch; however, due to the inherent variability in the many components under test, remote mode can result in larger timing variations, which can cause spurious test failures.

The packet plumbing is, naturally, a bit different in local and remote modes. To capture outgoing packets we use a packet socket (on Linux) or libpcap (on BSD-derived OSes). To inject packets locally we use a TUN device; to inject packets over the physical network in remote mode we again use a packet socket or libpcap. To consume test packets in local mode we use a TUN device; remotely, packets go over the physical network and the remote kernel drops them, because it has no interface with the test's remote IP address.

Local Mode

Local mode is the default, so to use it you need no special command line flags; you only need to provide the path of the script to execute:

```
./packetdrill foo.pkt
```

Remote Mode

To use remote mode, on the machine under test (the "client" machine), you must specify one command line option to enable remote mode (acting as a client) and then a second option to specify the IP address of the remote server machine to which the client packetdrill instance will connect. Only the client instance takes a packetdrill script argument, which can be the path of any ordinary packetdrill test script:

```
client# ./packetdrill --wire_client --wire_server_ip=<server_ip>
foo.pkt
```

On the remote machine, on the same layer 2 broadcast domain (e.g., same Ethernet switch), run the following to have a packetdrill process act as a "wire server" daemon to inject and sniff packets remotely on the wire:

```
server# ./packetdrill --wire_server
```

How does this work? First, the client instance connects to the server (using TCP), and sends the command line options and the contents of the script file. Then the two packetdrill instances work in concert to execute the script and test the client machine's network stack.

Model	Syntax Example	Description
Absolute	0.75	The specific time at which an event should occur.
Relative	+0.2	The interval after the last event at which an event should occur.
Wildcard	*	Allows an event to occur at any time.
Range	0.750~0.900	The absolute time range in which the event should occur.
Relative Range	+0.1~+0.2	The relative time range after the last event in which the event should occur.
Loose	--tolerance_usecs=800	Allows all events to happen within a range (from the command line).
Blocking	0.750...0.900	Specifies a blocking system call that starts/returns at the given times.

Table 1: Timing models supported by packetdrill

Timing Models

Because many protocols are sensitive to timing, we added support for significant timing flexibility in scripts. Each statement has a timestamp, enforced by packetdrill: if an event does not occur at the specified time, packetdrill flags an error and reports the actual time. Table 1 shows the packetdrill timing models.

Protocol Features

IPv4 and IPv6

packetdrill supports IPv4, IPv6, and dual-stack modes. The user specifies which mode to use when executing a test by using the `--ip_version` command line flag: AF_INET sockets with IPv4 traffic (`--ip_version=ipv4`), AF_INET6 sockets with IPv6 traffic (`--ip_version=ipv6`), and AF_INET6 sockets with IPv4 traffic (`--ip_version=ipv4-mapped-ipv6`).

To enable running the same script unmodified in any of the three modes, scripts omit IP-version-specific aspects of packets and system calls. For example, scripts do not specify the local and remote IP addresses of packets inside the script itself. Likewise, scripts do not specify a domain (AF_INET or AF_INET6) in a socket() call, nor do they specify the address and address length in a bind() call. As a result, getting a local test originally used for AF_INET sockets and IPv4 to work in other addressing modes is easy.

To run the test using AF_INET6 sockets with IPv4 traffic, use:

```
./packetdrill --ip_version=ipv4-mapped-ipv6 foo.pkt
```

To run the test using AF_INET6 sockets with IPv6 traffic, you'll need to specify both `--ip_version` and an MTU that is 20 bytes larger than the typical 1500-byte MTU, to accommodate the IPv6 header, which is 20 bytes larger than the IPv4 header:

```
./packetdrill --ip_version=ipv6 --mtu=1520 foo.pkt
```

With these small adjustments to the packetdrill command line, you can test all three addressing modes with a single script, with no extra development work.

Note that to get FreeBSD and NetBSD to allow using ipv4-mapped-ipv6 mode you must first tell the kernel you want to enable this mode of operation with:

```
sysctl -w net.inet6.ip6.v6only = 0
```

Also note that OpenBSD does not support ipv4-mapped-ipv6 mode because it explicitly disallows AF_INET6 sockets from handling IPv4 traffic.

Path MTU Discovery

packetdrill allows testing of Path MTU Discovery, which most TCP senders use to dynamically find an Internet path's maximum transmission unit (MTU), the biggest packet size that can safely traverse the path without suffering a performance hit due to IP-layer fragmentation and reassembly. Path MTU Discovery is described in RFC 1191 for IPv4 and RFC 1981 for IPv6. The basic idea is that senders mark the "Don't Fragment" (DF) bit in all outgoing IP headers. If a router along the path sees that it needs to fragment the packet but the DF bit is set, then the router sends an ICMP message saying "unreachable - fragmentation needed and DF set," with the MTU that the sender should use. When the sender receives this ICMP message, it retransmits any outstanding data and uses smaller packets in the future.

Listing 2 shows a simple Path MTU scenario (this script passes on Linux):

```
// Send a data segment.
+0 write(4, ..., 1460) = 1460
+0 > P. 1:1461(1460) ack 1

// ICMP says that segment was too big.
+0.100 < [1:1461(1460)] icmp unreachable frag_needed mtu 1200

// TCP retransmits with smaller packet size.
+0 > . 1:1161(1160) ack 1
+0 > P. 1161:1461(300) ack 1
```

Listing 2: TCP Path MTU Discovery example

Explicit Congestion Notification

packetdrill supports Explicit Congestion Notification, or ECN (see RFC 3168), a standard protocol that allows routers to explicitly signal to Internet transports (typically TCP) that there is congestion in the network by setting bits in the IP header. The ECN approach has several advantages over the traditional congestion signaling mechanism of dropping packets, but it is not yet widely deployed.

Any packet can have an ECN clause following the direction (< or >) field. Tests that do not care about ECN (and most tests do not) can simply omit the ECN clause. The supported ECN clauses allow tests to directly specify the injected or expected values of the two ECN bits; they are:

- [noecn] The IP ECN field is 00; sender transport (e.g., TCP) does not support ECN
- [ect1] The IP ECN field is 01, ECT(1), indicating "ECN-Capable Transport"
- [ect0] The IP ECN field is 10, ECT(0), indicating "ECN-Capable Transport"
- [ce] The IP ECN field is 11, set by a router to say "Congestion Experienced"

One interesting aspect of ECN is that ECN-capable senders (such as ECN-savvy TCP stacks) can set the ECN bits to either the ECT(0) or ECT(1) codepoints to indicate that they "speak ECN." This allows the sender and receiver to collaborate to detect whether some network element or receiver is corrupting or lying about the ECN bits, which would disrupt congestion signaling and potentially allow senders to grab an unfair share of bandwidth (see RFC 3540, Robust Explicit Congestion Notification (ECN) Signaling with Nonces). To cope with this potential variation, packetdrill also allows outgoing packets to use a fourth type of ECN clause, which specifies that an outgoing packet should have either the ECT(0) or ECT(1) codepoint:

- [ect01] The (outgoing) IP ECN field should be 10 or 01

Future Work

packetdrill can be used at present for testing not only fundamental network protocols that it supports natively (TCP, UDP, and ICMP on IPv4/IPv6) but also applications that use these protocols (e.g., a Web application that runs over TCP); however, because packetdrill has no knowledge of application-level datagrams, its ability to mimic, in script form, specific higher-layer protocols and application interactions is limited. We hope to make it easier for users to specify application-level payloads to be sent or received.

Also, packetdrill currently only supports testing a single connection at a time. We hope to extend it to support testing multiple concurrent connections. Furthermore, although packetdrill currently supports local (stand-alone) and on-the-wire (two-host) operations, it does not yet support multi-host operation or testing a remote machine that is not itself running packetdrill. These may be useful in some cases, and they should be straightforward to add to the current framework.

We welcome patches from the community, both for bug fixes and new features.

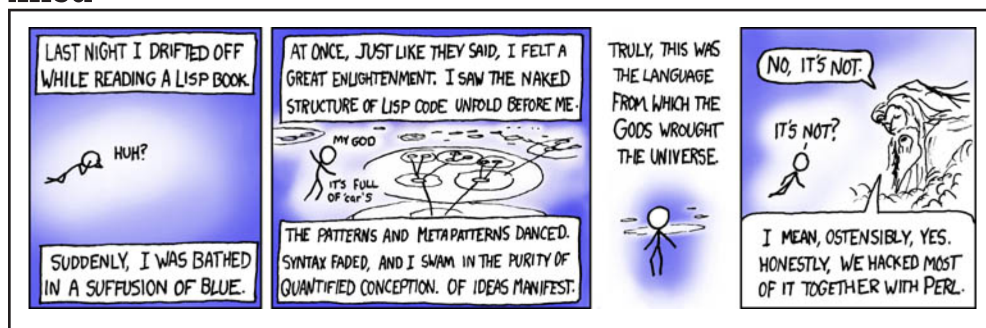
References

[1] Neal Cardwell, et al, "packetdrill: Scriptable Network Stack Testing, from Sockets to Packets," USENIX ATC 2013: https://www.usenix.org/conference/atc13/packetdrill-scriptable-network-stack-testing-sockets-packets.

[2] packetdrill open source project home and git repository: https://code.google.com/p/packetdrill/.

[3] packetdrill email list, for questions, discussion, and patches: http://groups.google.com/group/packetdrill.

xkcd



xkcd.com

Buy the Box Set!

Whether you had to miss a conference, or just didn't make it to all of the sessions, here's your chance to watch (and re-watch) the videos from your favorite USENIX events. Purchase the "Box Set," a USB drive containing the high-resolution videos from the te chnical sessions. This is perfect for folks on the go or those without consistent Internet access.

Box Sets are available for:

- **USENIX Security '13:** 22nd USENIX Security Symposium
- **HealthTech '13:** 2013 USENIX Workshop on Health Information Technologies
- **WOOT '13:** 7th USENIX Workshop on Offensive Technologies
- **UCMS '13:** 2013 USENIX Configuration Mangement Summit
- **HotStorage '13:** 5th USENIX Workshop on Hot Topics in Storage and File Systems
- **HotCloud '13:** 5th USENIX Workshop on Hot Topics in Cloud Computing
- **WiAC '13:** 2013 USENIX Women in Advanced Computing Summit
- **NSDI '13:** 10th USENIX Symposium on Networked Systems Design and Implementation
- **FAST '13:** 11th USENIX Conference on File and Storage Technologies
- **LISA '12:** 26th Large Installation System Administration Conference

Learn more at:
www.usenix.org/boxsets

COLUMNS

Practical Perl Tools

Parse Me, Amadeus

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

In past columns we've had the pleasure of looking at configuration file processing of all sorts. We've discussed ways to work with simple file formats like .ini files and more complex formats like XML, YAML, and JSON. But what if you find you need something even more sophisticated? What if you find you need a config that is actually a mini-language (some would call it a DSL, or domain specific language)? In cases like that you'll have to write code that can parse this language so your program can work with the directives you've specified. This column is about one of the more popular and more powerful modules for this work.

The Basics

I should note that when you start to say words like "parse" the computer scientists in the room perk up their ears because they've all had the pleasure of studying compiler design at some point in their academic career. I personally haven't cracked open the canonical but actually really good tome on the subject ("the Dragon Book," aka *Compilers: Principles, Techniques, and Tools*) in quite a few years. My apologies if I am playing faster and looser with terminology around parsing than perhaps I should as a graduate of that august field. But let's talk about a few key ideas before actually seeing any code. The key things I want to get into are the "how" and the "what" of the process. But warning: we're going to only skim the surface of all of the subjects mentioned in this column.

Typically, a parser's job is to take in a set of "tokens" and decide if it makes sense in terms of some language definition (and if it does, the parser hands the program back some sort of data structure that contains the results of the parse). Let's see a simple language so I can show you what I mean by token. Most parsing tutorials start out with a calculator example (the tokens are "numbers" and "operators" where one of the operations might be "plus"), but let's use something slightly more interesting:

```
recipe strawberry lemonade popsicle
ingredient frozen lemonade - 12 ounces
ingredient cold water - 3 cups
ingredient frozen sliced strawberries - 16 ounces
direction stir lemonade + water
direction blend strawberries
direction stir strawberries + lemonade
direction freeze
```

In this case, I could say the first line above was made up of "recipe" followed by a NAME. The second line has "ingredient" an ingredient NAME, and a QUANTITY. Later on we see "direction," an ACTION and a set of OBJECTS. All of these things can be considered tokens.

As a related aside (if just to satisfy some of the other CS majors who are jumping up and down on the sidelines with their hands in the air waiting to point this out): there is a process that takes place before parsing, namely changing the plain stream of incoming text to tokens (r..e..c..i..p..e..<space>.. gets turned into "recipe" which is a RECIPE_LABEL token). That is

typically handled by lexer code. The Perl module we'll be using has a lexer built in so we won't need to explicitly do much lexing, but it is good to know what is going on when we get to that point.

Grammars Rock

We just discussed a bit of the "how" parsing works; now let's look into the "what" we are parsing. We need a way to tell a parser "here's the definition of the language to parse." More often than not, that definition takes the form of a grammar. Here's a simple grammar that matches the recipe language example we see above:

```
NAME INGREDIENT+ DIRECTION+
NAME: 'recipe' name
INGREDIENT: 'ingredient' name '-' amount UNIT
UNIT: 'ounces' | 'cups' | 'pounds'
DIRECTION: 'direction' action | 'direction' action name '+' name
```

Let's walk through this grammar one line ("rule") at a time. The first rule says a line consists of a NAME line followed by one or more INGREDIENT lines and then by one or more DIRECTION lines. Subsequent rules define what those kinds of lines contain. A NAME line starts off with the literal string 'recipe' followed by the name of the recipe. An INGREDIENT line starts with 'ingredient' followed by the name, a literal dash, and the amount of the ingredient in one of several possible units (as specified in the subsequent rule). Finally, we provide a DIRECTION line that can either specify just an action or an action that takes place between two of the ingredients.

One thing that may be a bit surprising about this grammar is the first line. You might be tempted to write it like this (as I did at first when writing this article):

```
NAME | INGREDIENT+ | DIRECTION+
```

because it might seem like we'll be parsing a recipe name line or some number of ingredient lines, or some number of direction lines. And indeed, we will be parsing one of those kinds of lines at a time. But if we want to specify that we are parsing one of those, followed by the next, followed by the next thing, we won't be specifying them as alternatives. If we do, then the parser can say, "Okay, let's match the first rule. The first rule says I need to find just one of those alternatives from the list. Found one. Okay, that rule has matched so I must be done parsing." Instead, we say we'll need to say we expect one thing after another.

Bring on the Perl

Now that we have a grammar that specifies what we want to parse and a sample document to parse, let's put tab A into slot B. There are a number of Perl modules for parsing grammars, but the one we're going to look at is Parse::RecDescent. Parse::RecDescent has been around since 1997 and is one of the grand dames of the Perl parsing world at this point. We'll turn everything we've seen so far into a Perl program using that module:

```
use Parse::RecDescent;

my $grammar = q {
   startrule: recipename ingredient(s) direction(s)
   recipename: 'recipe' name
   ingredient: 'ingredient' name '-' amount unit
   unit: 'ounces'
        | 'cups'
        | 'pounds'
   direction: 'direction' action name '+' name
           | 'direction' action name
           | 'direction' action
   action: /\w+/
   amount: /\d+/
   name: /[a-zA-Z0-9 ]+/
};

my $heredoc = <<END;
recipe strawberry lemonade popsicle
ingredient frozen lemonade - 12 ounces
ingredient cold water - 3 cups
ingredient frozen sliced strawberries - 16 ounces
direction stir lemonade + water
direction blend strawberries
direction stir strawberries + lemonade
direction freeze
END

my $parser = new Parse::RecDescent($grammar);

print defined $parser->startrule($heredoc) ? 'OK' : 'NOT OK', "\n";
```

The major parts of this program are pretty simple: first we list the grammar we're going to use (more on this in a moment), followed by the sample document we're going to parse. We request a Parse::RecDescent object that we next used to start the parse at the rule marked 'startrule' and perform a parse, printing the results.

Now that we're looking at actual code (finally!) it would probably be useful to compare the code to the previous grammar in our text because the differences will be illustrative. The first difference is our first line gets marked "startrule" so we know where to begin a parse. It would be reasonable to have a convention that a parse starts at the first rule listed, but no such convention exists for the module. This makes more sense if there could be two potential starting places for a parse, for example a "debug rule" and the real "start rule." The only problem with this explanation is I'm making this reason up. I've never seen people actually do this, but it sure sounds plausible, doesn't it?

```
 1 |startrule   |Trying subrule: [recipename]                                         |
 2 |recipename |Trying rule: [recipename]                                            |
 2 |recipename |Trying production: ['recipe' name]                                   |
 2 |recipename |Trying terminal: ['recipe']                                          |
 2 |recipename |>>Matched terminal<< (return value: [recipe])                         |
 2 |recipename |                                                                      |
 2 |recipename |                                                                      |" strawberry lemonade
   |           |                                                                      |popsicle\ningredient frozen
 2 |recipename |                                                                      |lemonade - 12
   |           |                                                                      |ounces\ningredient cold
 2 |recipename |                                                                      |water - 3 cups\ningredient
   |           |                                                                      |frozen sliced strawberries -
 2 |recipename |                                                                      |16 ounces\ndirection stir
   |           |                                                                      |lemonade + water\ndirection
 2 |recipename |                                                                      |blend
   |           |                                                                      |strawberries\ndirection stir
 2 |recipename |                                                                      |strawberries +
   |           |                                                                      |lemonade\ndirection
 2 |recipename |                                                                      |freeze\n"
 2 |recipename |Trying subrule: [name]                                               |
 3 |   name    |Trying rule: [name]                                                   |
 3 |   name    |Trying production: [/[a-zA-Z0-9 ]+/]                                  |
 3 |   name    |Trying terminal: [/[a-zA-Z0-9 ]+/]                                    |
 3|    name    |>>Matched terminal<< (return value: [strawberry lemonade popsicle])    |
 3 |   name    |                                                                      |
 3 |   name    |                                                                      |"\ningredient frozen
   |           |                                                                      |lemonade - 12
 3 |   name    |                                                                      |ounces\ningredient cold
   |           |                                                                      |water - 3 cups\ningredient
 3 |   name    |                                                                      |frozen sliced strawberries -
   |           |                                                                      |16 ounces\ndirection stir
 3 |   name    |                                                                      |lemonade + water\ndirection
   |           |                                                                      |blend
 3 |   name    |                                                                      |strawberries\ndirection stir
   |           |                                                                      |strawberries +
 3 |   name    |                                                                      |lemonade\ndirection
   |           |                                                                      |freeze\n"
 3 |   name    |                                                                      |
 3 |   name    |>>Matched production: [/[a-zA-Z0-9]+/]<<                               |
 3 |   name    |                                                                      |
 3 |   name    |>>Matched rule<< (return value: [strawberry lemonade popsicle])        |
 3 |   name    |                                                                      |
 3 |   name    |(consumed: [strawberry lemonade popsicle])                             |
 3 |   name    |                                                                      |
 2 |recipename |>>Matched subrule: [name]<< (return value: [strawberry lemonade popsicle] |
 2 |recipename |                                                                      |
 2 |recipename |>>Matched production: ['recipe' name]<<                               |
 2 |recipename |                                                                      |
 2 |recipename |>>Matched rule<< (return value: [strawberry lemonade popsicle])        |
 2 |recipename |                                                                      |
 2 |recipename |(consumed: [recipe strawberry lemonade popsicle])                      |
 2 |recipename |                                                                      |
```

Figure 1: If you set $::RD_TRACE variable in Parse::RecDescent to 1, you will get debugging output like this when parsing the first line in our example.

The second and perhaps more important difference between the grammar versions is the stuff at the bottom of the grammar. In the first appearance of our grammar we mentioned things like “name,” “amount,” and “action” without ever saying just what those things are (or more importantly, how the parser would know one if it bumped into one in a dark alley). If we left out those lines from our grammar, our program would throw the following errors:

```
Warning: Undefined (sub)rule "action" used in a production.
Warning: Undefined (sub)rule "name" used in a production.
Warning: Undefined (sub)rule "name" used in a production.
Warning: Undefined (sub)rule "name" used in a production.
Warning: Undefined (sub)rule "amount" used in a production.
```

Parse::RecDescent makes defining these parts of the grammar easy; we just need to provide a Perl regular expression that will match that part. We say a name can be a letter/number (plus a space if desired), an action is a single word, and an amount is one or more digits. And, yes, we are actually providing direction to the Parse::RecDescent lexer so it knows how to construct those tokens.

One last thing to point out is that Parse::RecDescent has a very legible (for English speakers) way of saying, “One or more of these rules.” We see that in action in the grammar where it uses this English pluralization idiom when it mentions “ingredient(s)” and “direction(s)” to indicate it is standing in for one or more of those things.

With all of this build up, what happens if we run this program? It outputs (oh, the suspense is delicious):

```
OK
```

If we changed the sample document so it said:

```
ingredient frozen lemonade - 12 bounces
```

it would print:

```
NOT OK
```

instead. Okay, maybe not so exciting, but actually this is useful. Now you know how to write a program that validates a document based on your mini-language. We’ll see how to actually capture the info in the document in just a second. Before we do, I want to mention a super- helpful Parse::RecDescent feature that you may find yourself using during development. If you add the following line to your code:

```
$::RD_TRACE = 1;
```

it spits out a ton of really useful debugging information about the parse. In the interest of space, let me show you a very small excerpt of the debug output.

In Figure 1, you can see the parse began with its start rule trying to match the subrule about the recipe name. The rule it is trying to match is found in the second column. In the trace in Figure 1, we can see that the parser looks for the literal string ‘recipe’, finds it, and then sees whether it can find the input it needs to collect a recipe name from the input it has available (shown in the third column). It succeeds, showing you the result of the matches and what part of the input it was able to consume.

So how do we use the information that Parse::RecDescent presumably could gather as it parses merrily along? To do that we have to discuss what the module calls “actions.” With Parse::RecDescent, you can specify what should happen at each step in the parse. For example, you might want to have the parser return the values it matched along the way so you can construct a data structure that the rest of your program will traverse. The simplest way to get into the action game is to use a feature called autoactions that lets you set a single action to automatically take place after every rule has been parsed. It gets specified something like this:

```
$::RD_AUTOACTION = q { [@item] };
```

(or you can sneak it into the grammar itself using a special tag). The @item array in an action holds info on the items that are being matched ($item[0] is the actual name of the rule that is being matched; the rest of the array specifies the other parts of what is found). There are other magic variables that can be referenced; see the doc for more information. If we took our previous program and added that autoaction line (plus loading Data::Dumper) and said instead:

```
my $parserresults = $parser->startrule($heredoc);

print Dumper $parserresults,"\n";
```

we would see output that began this way:

```
$VAR1 = [
          'startrule',
          [
            'recipename',
            'recipe',
            [
              'name',
              'strawberry lemonade popsicle'
            ]
          ],
          [
            [
              'ingredient',
              'ingredient',
              [
                'name',
                'frozen lemonade '
```

```
                  ],
                  '-',
                  [
                    'amount',
                    '12'
                  ],
                  [
                    'unit',
                    'ounces'
                  ]
                ],
                ...
```

For a more complex but precise parse tree, we can slip an <autotree> tag ahead of the startrule in the grammar, and Parse::RecDescent will create a data structure that begins like this:

```
$VAR1 = bless( {
  '__RULE__' => 'startrule',
  'recipename' => bless( {
      '__RULE__' => 'recipename',
      'name' => bless( {
                          '__VALUE__' => 'strawberry lemonade
                                popsicle'
                      }, 'name' ),
     '__STRING1__' => 'recipe'
  }, 'recipename' ),
  'ingredient(s)' => [
      bless( {
               'unit' => bless( {
                                   '__VALUE__' => 'ounces'
                                 }, 'unit' ),
               'amount' => bless( {
                                      '__VALUE__' => '12'
                                    }, 'amount' ),
               '__STRING2__' => '-',
               '__RULE__' => 'ingredient',
               'name' => bless( {
                                    '__VALUE__' => 'frozen
                                          lemonade '
                                  }, 'name' ),
                '__STRING1__' => 'ingredient'
               }, 'ingredient' ),
             ...
```

Now, what you do with that data structure once you get it is truly up to you. In our case, you could have something that engages your fully automated kitchen to make a popsicle for you.

I want to leave you pondering this little bit of free will, but before I go I think I would be remiss if I didn't mention that there are other really cool parsing modules available. The two that I have my eye on in particular are the Regexp::Grammars module (builds on the super-powerful regexp constructs in Perl 5.10+) and the Marpa::R2 module, which uses a very different parsing algorithm than Parse::RecDescent and can do some cool stuff that Parse::RecDescent can't. Do check them both out if parsing is in your future.

Take care and I'll see you next time.

Python: With That
Five Easy Context Managers

DAVID BEAZLEY

David Beazley is an open source developer and author of the *Python Essential Reference* (4th Edition, Addison-Wesley, 2009) and *Python Cookbook* (3rd Edition, O'Reilly Media, 2013). He is also known as the creator of Swig (http://www.swig.org) and Python Lex-Yacc (http://www.dabeaz.com/ply.html). Beazley is based in Chicago, where he also teaches a variety of Python courses.
dave@dabeaz.com

At the last PyCon conference, Raymond Hettinger gave a keynote talk in which he noted that context managers might be one of Python's most powerful yet underappreciated features. In case you're new to the concept of a context manager, we're talking about the `with` statement that was added to Python 2.6. You'll most often see it used in the context of file I/O. For instance, this is the "modern" style of reading a file line-by-line:

```
with open('data.csv') as f:
    for line in f:
        # Do something with line
        ...

# f automatically closed here
```

In this example, the variable `f` holds an open file instance that is automatically closed when control leaves the block of statements under the `with` statement. Thus, you don't have to invoke `f.close()` explicitly when you use the `with` statement as shown. If you're not quite convinced, you can also try an interactive example:

```
>>> with open('/etc/passwd') as f:
...     print(f)
...
<open file '/etc/passwd', mode 'r' at 0x2b4180>
>>> print(f)
<closed file '/etc/passwd', mode 'r' at 0x2b4180>
>>>
```

With that in mind, seeing how something so minor could be one of the language's most powerful features as claimed might be a bit of a stretch. So, in this article, we'll simply take a look at some examples involving context managers and see that so much more is possible.

Make a Sandwich

What is a context manager anyways? To steal an analogy from Raymond Hettinger, a context manager is kind of like the slices of bread that make up a sandwich. That is, you have a top and a bottom piece, in-between which you put some kind of filling. The choice of filling is immaterial—the bread doesn't pass judgment on your dubious choice to make a sandwich filled with peanut-butter, jelly, and tuna.

In terms of programming, a context manager allows you to write code that wraps around the execution of a block of statements. To make it work, objects must implement a specific protocol, as shown here:

```
class Manager(object):
    def __enter__(self):
        print('Entering')
        return "SomeValue"    # Can return anything
    def __exit__(self, e_ty, e_val, e_tb):
        if e_ty is not None:
            print('exception %s occurred' % e_ty)
        print('Exiting')
```

Before proceeding, try the code yourself:

```
>>> m = Manager()
>>> with m as val:
...     print('Hello World')
...     print(val)
...
Entering
Hello World
SomeValue
Exiting
>>>
```

Notice how the "Entering" and "Exiting" messages get wrapped around the statements under the `with`. Also observe how the value returned by the `__enter__()` method is placed into the variable name given with the optional as specifier. Now, try an example with an error:

```
>>> with m:
...     print('About to die')
...     x = int('not a number')
...
Entering
About to die
exception <class 'ValueError'> occurred
Exiting
Traceback (most recent call last):
  File "<stdin>", line 3, in
ValueError: invalid literal for int() with base 10: 'not a number'
>>>
```

Here, carefully observe that the `__exit__()` method was invoked and presented with the type, value, and traceback of the pending exception. This occurred prior to the traceback being generated.

You can make any object work as a context manager by implementing the `__enter__()` and `__exit__()` methods as shown; however, the contextlib library provides a decorator that can also be used to write context managers in the form of a simple generator function. For example:

```
from contextlib import contextmanager

@contextmanager
def manager():
    # Everything before yield is part of __enter__
    print("Entering")
    try:
        yield "SomeValue"
    # Everything beyond the yield is part of __exit__
    except Exception as e:
        print("An error occurred: %s" % e)
        raise
    else:
        print("No errors occurred")
```

If you try the above function, you'll see that it works in the same way.

```
>>> with manager() as val:
...     print("Hello World")
...     print(val)
...
Entering
Hello World
SomeValue
No errors occurred
>>>
```

Sandwiches Everywhere!

Once you've seen your first sandwich, you'll quickly realize that they are everywhere! Consider some of the following common programming patterns:

```
# File I/O
f = open('somefile')
...
f.close()

# Temporary files/directories
name = mktemp()
...
remove(name)

# Timing
start_time = time()
...
end_time = time()

# Locks (threads)
lock.acquire()
...
lock.release()
```

```
# Publish-subscribe
channel.subscribe(recipient)
...
channel.unsubscribe(recipient)

# Database transactions
cur = db.cursor()
...
db.commit()
```

Indeed, the same pattern repeats itself over and over again in all sorts of real-world code. In fact, any time you find yourself working with code that follows this general pattern, consider the use of a context manager instead. Indeed, many of Python's built-in objects already support it. For example:

```
# File I/O
with open('somefile') as f:
    ...

# Temporary files
from tempfile import NamedTemporaryFile
with NamedTemporaryFile() as f:
    ...

# Locks
lock = threading.Lock()
with lock:
    ...
```

The main benefit of using the context-manager version is that it more precisely defines your usage of some resource and is less error prone should you forget to perform the final step (e.g., closing a file, releasing a lock, etc.).

Making Your Own Managers

Although it's probably most common to use the `with` statement with existing objects in the library, you shouldn't shy away from making your own context managers. In fact, it's pretty easy to write custom context manager code.

The remainder of this article simply presents some different examples of custom context managers in action. It turns out that they can be used for so much more than simple resource management if you use your imagination. The examples are presented with little in the way of discussion, so you'll need to enter the code and play around with them yourself.

Temporary Directories with Automatic Deletion

Sometimes you need to create a temporary directory to perform a bunch of file operations. Here's a context manager that does just that, but it takes care of destroying the directory contents when done:

```
import tempfile
import shutil
from contextlib import contextmanager

@contextmanager
def tempdir():
    name = tempfile.mkdtemp()
    try:
        yield name
    finally:
        shutil.rmtree(name)
```

To use it, you would write code like this:

```
with tempdir() as dirname:
     # Create files and perform operations
     filename = os.path.join(dirname, 'example.txt')
     with open(filename, 'w') as f:
         f.write('Hello World\n')
     ...

# dirname (and all contents) automatically deleted here
```

Ignoring Exceptions

Sometimes you just want to ignore an exception. Traditionally, you might write code like this:

```
try:
    ...
except SomeError:
    pass
```

However, here's a context manager that allows you to reduce it all to one line:

```
@contextmanager
def ignore(exc):
    try:
        yield
    except exc:
        pass

# Example use. Parse data and ignore bad conversions
records = []
for row in lines:
    with ignore(ValueError):
         record = (int(row[0]), int(row[1]), float(row[2]))
         records.append(record)
```

With a few minor modifications, you could adapt this code to perform other kinds of exception handling actions: for example, routing exceptions to a log file, or simply packaging up a complex exception handling block into a simple function that can be easily reused as needed.

Making a Stopwatch

Here's an object that implements a simple stopwatch:

```
import time

class Timer(object):
    def __init__(self):
        self.elapsed = 0.0
        self._start = None

    def __enter__(self):
        assert self._start is None, "Timer already started"
        self._start = time.time()

    def __exit__(self, e_ty, e_val, e_tb):
        assert self._start is not None, "Timer not started"
        end = time.time()
        self.elapsed += end - self._start
        self._start = None

    def reset(self):
        self.__init__()
```

To use the timer, you simply use the `with` statement to indicate the operations you want timed. For example:

```
# Example use
my_timer = Timer()

...
with my_timer:
     statement
     statement
     ...
...
print("Total time: %s" % my_timer.elapsed)
```

Deadlock Avoidance

A common problem in threaded programs is deadlock arising from the use of too many locks at once. Here is a context manager that implements a simple deadlock avoidance scheme that can be used to acquire multiple locks at once. It works by simply forcing multiple locks always to be acquired in ascending order of their object IDs.

```
from contextlib import contextmanager

@contextmanager
def acquire(*locks):
    sorted_locks = sorted(locks, key=id)
    for lock in sorted_locks:
        lock.acquire()
    try:
        yield
    finally:
        for lock in reversed(sorted_locks):
            lock.release()
```

This one might take a bit of pondering, but if you throw it at the classic "Dining Philosopher's" problem from operating systems, you'll find that it works.

```
import threading

def philosopher(n, left_stick, right_stick):
    while True:
        with acquire(left_stick, right_stick):
            print("%d eating" % n)

def dining_philosophers():
    sticks = [ threading.Lock() for n in range(5) ]
    for n in range(5):
        left_stick = sticks[n]
        right_stick = sticks[(n + 1) % 5]
        t = threading.Thread(target=philosopher,
                             args=(n, left_stick, right_stick))
        t.daemon = True
        t.start()

if __name__ == '__main__':
    import time
    dining_philosophers()
    time.sleep(10)
```

If you run the above code, you should see all of the philosophers running deadlock free for about 10 seconds. After that, the program simply terminates.

Making a Temporary Patch to Module

Here's a context manager that allows you to make a temporary patch to a variable defined in an already loaded module:

```
from contextlib import contextmanager
import sys

@contextmanager
def patch(qualname, newvalue):
    parts = qualname.split('.')
    assert len(parts) > 1, "Must use fully qualified name"
    obj = sys.modules[parts[0]]
    for part in parts[1:-1]:
        obj = getattr(obj, part)

    name = parts[-1]
    oldvalue = getattr(obj, name)
    try:
```

```
        setattr(obj, name, newvalue)
        yield newvalue
    finally:
        setattr(obj, name, oldvalue)
```

Here's an example of using this manager:

```
>>> import io
>>> with patch('sys.stdout', io.StringIO()) as out:
...     for i in range(10):
...         print(i)
...
>>> out.getvalue()
'0\n1\n2\n3\n4\n5\n6\n7\n8\n9\n'
>>>
```

In this example, the value of `sys.stdout` is temporarily replaced by a `StringIO` object that allows you to capture output directed toward standard output. This might be useful in the context of certain tasks such as tests. In fact, the popular mock tool (https://pypi.python.org/pypi/mock) has a similar, but much more powerful variant of this decorator.

More Information

This article is really only scratching the surface of what's possible with context managers; however, the key takeaway is that context managers can be used to address a wide variety of problems that come up in real-world programming. Not only that, they are relatively easy to define, so you're definitely not limited to using them only with Python's built-in objects such as files. For more ideas and inspiration, a good starting point might be documentation for the `contextlib` module as well as PEP 343 (http://www.python.org/dev/peps/pep-0343/).

BECOME A USENIX SUPPORTER AND REACH YOUR TARGET AUDIENCE

The USENIX Association welcomes industrial sponsorship and offers custom packages to help you promote your organization, programs, and products to our membership and conference attendees.

Whether you are interested in sales, recruiting top talent, or branding to a highly targeted audience, we offer key outreach for our sponsors. To learn more about becoming a USENIX Supporter, as well as our multiple conference sponsorship packages, please contact sponsorship@usenix.org.

Your support of the USENIX Association furthers our goal of fostering technical excellence and innovation in neutral forums. Sponsorship of USENIX keeps our conferences affordable for all and supports scholarships for students, equal representation of women and minorities in the computing research community, and the development of open source technology.

www.usenix.org/usenix-corporate-supporter-program

iVoyeur
Hearsay Among Monitoring Systems

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

This may be a little premature to talk about, but lately I've been consumed by an idea that is conceptually rooted in the complexity involved in making monitoring systems talk to each other. For someone who writes articles about making monitoring systems talk to each other, this is perhaps natural, but I know I'm not the only one who has noticed that adding a new monitoring system to an existing infrastructure does not linearly increase its complexity.

For example, say you have Nagios and want to add Splunk, and you want them to talk to each other, feeding passive check results from Splunk to Nagios and also round-trip times for an HTTP service in Nagios into Splunk. Then you add Ganglia and Collectd to the mix in a similar fashion. This scenario, depicted in Figure 1, begets four custom configurations for Nagios alone, one for Nagios itself, one for Nagios to talk to Splunk, another for Nagios to talk to Collectd, and yet another for Nagios to talk to Ganglia. Some of these systems will need to be configured in kind to talk back to Nagios.

I/O Hooks Aren't Enough Anymore

So inter-system configuration complexity is something like $(n-x)^2+(nx)$, where x is the number of send or receive-only, Graphite/Collectd-style tools you plug in to your monitoring architecture. If we were talking algorithms, we'd reduce this to $O(n^2)$ and be done. Effective systems monitoring requires a toolbox, but every tool you add to the box means reconfiguring all tools.

This complexity is obviously a hassle, but worse, it has a tendency to make snowflakes of your monitoring systems, eventually resulting in highly customized, fragile infrastructure. The alternative is to limit our visibility by forgoing the use of good tools to avoid the configuration burden (or installing them as stand-alone). A nearly exponential increase in configuration complexity makes this a hard limit for everyone, which is to say every shop WILL have to pick and choose a few tools from an increasingly huge list of amazingly great monitoring systems if they want them to work together.

At the risk of sounding melodramatic, I am saddened by this. I want all of these great monitoring tools to work like Legos. I want to plug them in to each other and build things with them. I want them to play to each other's strengths and become more than the sum of their parts.

There's No I in "Common Data Model"

Imagine for a moment that instead of each system having its own unique I/O hooks, they all supported a common data interchange format. If they all just woke up one morning and agreed to send and receive the same format messages. As depicted in Figure 2, they would no longer need to be configured specifically to communicate to each other, and could instead each be configured simply to enable import and/or export of the common format. Each

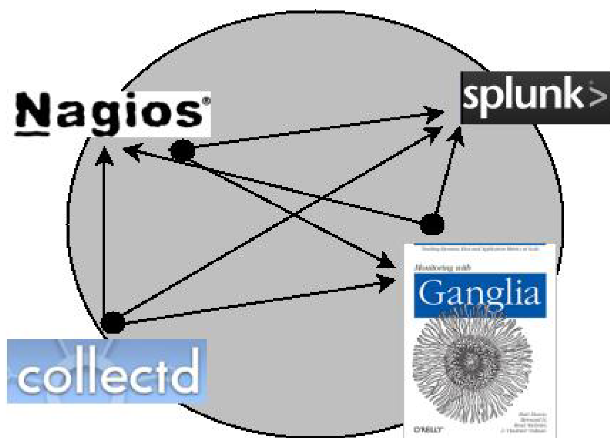


Figure 1: Each system must be custom configured for interoperability with the others.

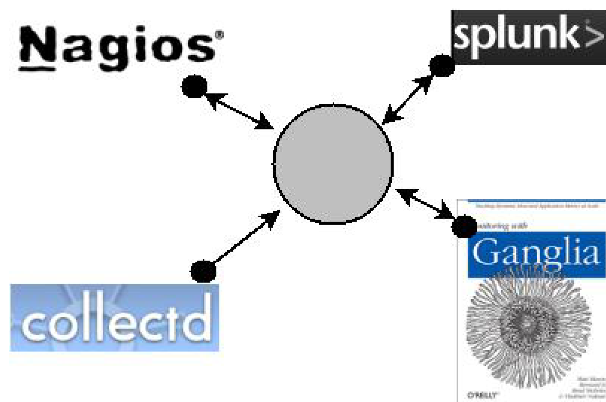


Figure 2: Each system merely enables support for a common data model.

monitoring system could share data out in a system-agnostic way, and other systems could pick and choose the state and metric data that was relevant to them regardless of the source.

This would greatly reduce the cost of adding new monitoring infrastructure, and would make everyone's life easier. But is it even possible to translate the output of every monitoring system to a common format that works as the input of every other?

Although it seems unlikely, practically speaking, all monitoring systems deal with similar data. The Riemann Project's event type, described at [1], characterizes a system-agnostic blob of monitoring data pretty perfectly. Copied directly from that site, the structure looks like this:

host	A hostname, e.g., "api1", "foo.com"
service	e.g., "API port 8000 reqs/sec"
state	Any string less than 255 bytes, e.g., "ok", "warning", "critical"
time	The time of the event, in UNIX epoch seconds
description	Freeform text
tags	Freeform list of strings, e.g., ["rate", "fooproduct", "transient"]
metric	A number associated with this event, e.g., the number of reqs/sec.
ttl	A floating-point time, in seconds the event is valid for

Every monitoring system I've worked with generates data that fits pretty well into this struct, and most fit with room to spare. Formalizing this, changing the "state" field to a Nagios-style int, and adding a UID field to make it possible to sign the messages and/or provide a unique hash so that they can be more easily de-duplicated/commuted etc. produces my own definitions:

string	Host //hostname, e.g., "foo.com",
string	Service //e.g., "HTTP reqs/sec"
uint8	State //Nagios style 0 ok, 1 warn, 2 crit, 3 unk 4-10 reserved
time_t	Time //the time the event occurred
string	Description //non-numeric state, event, or service description
string[]	Tags //list of tags, e.g., ["sentby:alice","src:nagios"]
float64	Metric //a metric, e.g., the number of reqs/sec.
uint32	TTL //valid time-to-live (in seconds) for this message
string	UID //unique hash or signature from host+service+time+State+Metric

Okay, Let's Kick This Pig

In a perfect world, I could at this point assemble the minions, kidnap the maintainer of every monitoring system, and demand that they import and export this structure for all the relevant events their systems generate. But despite my lack of minions, other problems need solving first, beginning with who pushes and who pulls, and continuing on through wire encoding (protobuf? JSON? XML? etc.), and the litany of details associated with actually putting the messages on the wire, routing them to where they need to go, and figuring out what to do when they get there. So I think, before I can push for native adoption, that there will need to be a fairly well developed model for how data exchange should operate in practice. We need to see what it looks like before we can decide whether it's worth doing.

To that end, libhearsay is a library that implements this common data format and comes with a couple of tools to simplify the protocol and data exchange details. Written in Golang [2] over the past few weeks when I should have been washing the dishes, libhearsay tools employ JSON and Zeromq [3] (sometimes written as 0MQ) to distribute "scraps" of hearsay between monitoring

systems (spewers and listeners), enabling your monitoring systems to gossip to each other.

Any monitoring system with something to share can be made into a hearsay spewer with the "spewer" utility. Spewer reads JSON-formatted scraps of hearsay from STDIN or a FIFO, verifies that they are valid (requiring either a host or service name, and a metric or state value), and puts them on the wire via Zeromq push or pub. In push mode, Zeromq will fan out the scraps by fairly distributing them among the connected listeners. In pub(lish) mode, Zeromq will broadcast each scrap to every subscriber.

The generic listener listens to a comma-separated list of spewer socket addresses, and outputs JSON-encoded scraps to STDOUT or appends them to a file of your choosing. The generic listener also has a "Nagios" mode that injects passive check results directly into your Nagios CMD file. Inter-system compatibility will be achieved through the creation of many task-specific listeners that are designed to work with specific tools such as Munin, Reimann, Zabbix, Zenoss, Reconnoiter, Graphite, etc. Each of these listeners will "just work," meaning that given the address of a spewer or several spewers, they'll take scraps off the wire, validate them, and inject them into their parent monitoring system in the way that system expects to receive them.

For now, the generic_listener and some shell scripts can help us get by, and hopefully prove the model, but step 2 certainly centers around the creation of a litany of purpose-specific listeners (some of which should be written by the time you read this). At that point, the cost of entry will be low enough that "normal users" will be able to play. Step 3 will be to push for native support. If you're a project maintainer, expect to see me at your con next year.

Patterns

Zeromq subscribers provide a filter when they subscribe to a pub socket, which enables them to discard the messages they aren't interested in. This should work handily with the "Tag" field in our scrap struct. The model I have in mind for my shop looks pretty much like Figure 2, where all spewers and listeners connect to a central set of redundant message brokers and use filters to extract the scraps from the systems they're interested in.

These brokers are nothing more than a set of systems that have both a listener (to accept scraps from every monitoring server) and a spewer (to copy every scrap back to the interested listeners). Something like a Brooklyn barber shop, all systems know to go to these hosts to both share and receive new hearsay. I imagine that each spewer will use the spewer utilities' "-t" switch to add a tag to each scrap they send, identifying it as, for example: "src:nagios", and each listener will filter for tags of this or that type.

Interestingly, given just the generic spewer and listener tools, any sort of distributed message-passing architecture could be built, and although I'm excited about the possibility of my "smorgasbord of monitoring data" model, I'm even more intrigued to see what other admins might design.

Wait, how does this work exactly?

Let's take a look at the spewer tool in practice by launching it with "-d" to trigger debug mode and sending it a partial scrap like so:

```
[dave@vlasov]--> echo '{"Host":"foo.com","Service":"HTTP",
"State":0}' | spewer -d
Starting Server
got message: {"Host":"foo.com","Service":"HTTP","State":0}
Sending:
{"Host":"foo.com","Service":"HTTP","State":
0,"Time":"2013-07-26T13:51:47.277299512-05:00","Description":""
,"Tags":["Spewed-by:
vlasov.dbg.com"], "Metric":-42,"TTL":60,"UID":""}
```

As you can see, given only a hostname, service name, and state value, spewer created a full scrap by populating default values for Time, Metric, and TTL, and adding a "Spewed-by:" tag, which should help us avoid message loops in the future. If I'd given spewer a "-u" switch, it would have generated an MD5 hash-sum of the message and assigned it to UID.

Spewer also created a 0MQ push socket and placed the scrap on the wire for any connected listeners. If we had a generic listener connected to localhost port 5000, spewer would have read the message and printed it back to STDOUT. If five listeners had been listening, 0MQ would have (round-robin) distributed the message to one of them. If I'd specified "-m pub", spewer would have opened a pub socket and every one of the connected five listeners would have gotten its own copy of the message.

There are myriad ways to get data out of Nagios and into the spewer, but I haven't made a final decision on what interim Nagios support looks like exactly. Because Nagios provides handy macros for things such as hostname, service name, and state, I'm tempted to write a little tool that is intended to be called from a notification command that could inject a scrap into spewer, or modify spewer to accept incoming scraps on a TCP socket locally.

Spewer cannot itself be called via a Nagios command because it needs to persist the publisher socket, and therefore must run as a daemon-like entity. Other options are a Nagios Event Broker module that could inject scraps into spewer, or something as simple as a shell script that could tail a performance log file from Nagios, translating and providing scraps to spewer via STDIN. Each approach has pros and cons.

I also anticipate the need for an indexing service of some sort to enable listeners to find spewers securely. I'll cross that bridge when I come to it.

This may be a long road to a dead-end, but at the moment I'm optimistic and by the next issue expect to have some real systems talking to each other. If you'd like to hack along, feel free to grab libhearsay from GitHub [4] or my blog [5]. Any help would be vastly appreciated and is 100% guaranteed to be repaid in beer at the first convenient conference we both attend.

Take it easy.

References

[1] http://riemann.io/concepts.html.

[2] http://golang.org/.

[3] http://www.zeromq.org.

[4] https://github.com/djosephsen/Hearsay.

[5] http://www.skeptech.org/hearsay.

SAVE THE DATE!

nsdi '14

11th USENIX Symposium on Networked Systems Design and Implementation

APRIL 2–4, 2014 • SEATTLE, WA

Join us in Seattle, WA, April 2-4, 2014, for the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI '14). NSDI focuses on the design principles, implementation, and practical evaluation of networked and distributed systems. Our goal is to bring together researchers from across the networking and systems community to foster a broad approach to addressing overlapping research challenges.

Program Co-Chairs: Ratul Mahajan, *Microsoft Research,* and Ion Stoica, *University of California, Berkeley*

www.usenix.org/conference/nsdi14

For Good Measure
Trending North

DAN GEER AND MUKUL PAREEK

Dan Geer is the CISO for In-Q-Tel and a security researcher with a quantitative bent. He has a long history with the USENIX Association, including officer positions, program committees, etc. dan@geer.org

Mukul Pareek is a risk management professional based in New York. Mukul is the co-publisher of the *Index of Cyber Security* and the author of a risk education Web site, riskprep.com. mp@pareek.org

> Trends, like horses, are easier to ride in the direction they are going.
>
> — *John Naisbitt*

We are the operators of the Index of Cyber Security (ICS), a two-and-a-half-year-old effort to inform the community at large of the trends in cybersecurity, which is to say the components of risk that the digital world brings with it. Monthly, we poll people with operational responsibility for cybersecurity on how two dozen different cybersecurity risks have changed in the past month, and, from that polling, calculate the ICS. Our Web site [1] has details. The questions are each five-point Likert scales, as in this example:

> Compared to last month, the threat from mass malware has
> fallen fast, fallen, stayed static, risen, risen fast

The ICS is a risk index; it is a time-series snapshot of cybersecurity risk as collectively seen by vetted, front-line practitioners. The ICS does not model the world but rather observes, in a structured way, the state of play in the cybersecurity space. If the ICS rises, it is because the respondents as a group have seen risk rise in their own work across the aggregate of all two dozen vectors of risk.

For the general public, we publish the ICS in the same way that the Conference Board publishes the Consumer Confidence Index [2], as a single number at the end of the month. For our respondent group, we publish monthly detailed reports and an Annual Report. This column excerpts the most recent Annual Report, for the year ending March 2013.

Cybersecurity risk has been rising since inception though the month-over-month change has varied, as seen in Figure 1.

Perhaps more to the point, the rate of increase is itself increasing, as seen either by comparing the compound annual growth rate (CAGR) since inception of the index to the CAGR calculated only over the most recent six months (in Figure 2), or as seen by normalizing the monthly changes in the ICS to their collective median and noting the distribution of divergence from that median (in Figure 3).

This finding of not just a continuing rise but an accelerating rise in the aggregate cybersecurity risk, as seen by front-line practitioners, has decision support value both to other practitioners and to policy makers.

The wide digital community is spending more time and treasure on cybersecurity every month as well, that is, the race between "Are we getting better?" (yes) and "Is the attackable

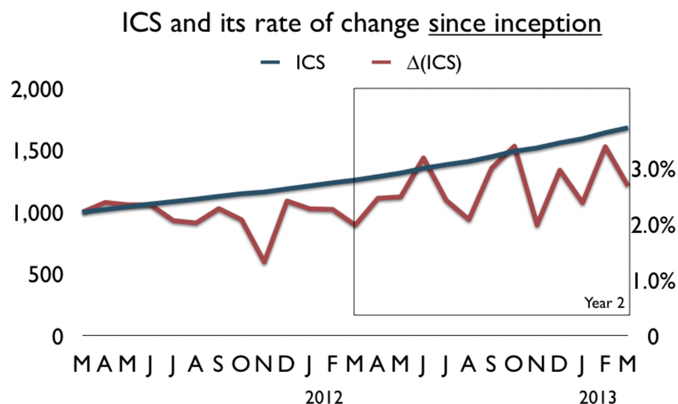


Figure 1: Index of Cyber Security and its rate of change

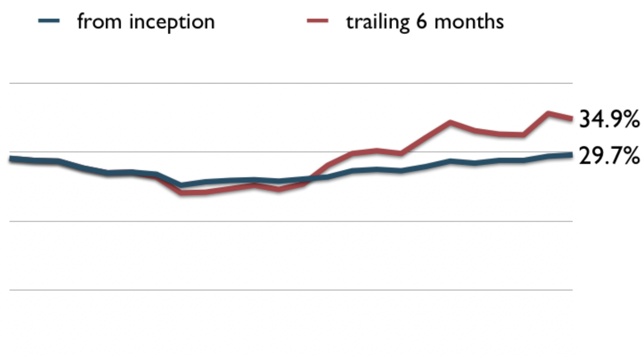


Figure 2: Compound annual growth rate since inception vs the most recent six months

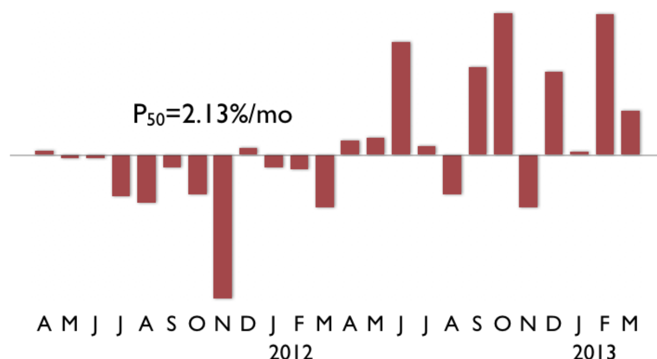


Figure 3: Rate of change of the ICS normalized to the median

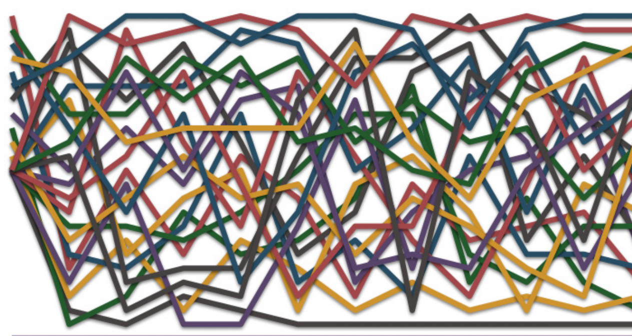


Figure 4: Tracing the rank of component risks month-to-month

surface growing?" (yes again) is a race being won by attackability. Cue the Red Queen: "It takes all the running you can do to keep in the same place."

Each question we ask has its own trendline, and each question is a component risk of the overall ICS. Now you might at first think that some component risk dominates the overall ICS, thus reducing the ICS to a noisy measure of that one risk. You'd be wrong. Rank ordering the spot (month over month) change in component risk contribution to the overall ICS is simply the rat's nest in Figure 4.

If you look not at the month over month (spot) impact of individual risks on the ICS in sum but rather at the cumulative impact of those individual risks, the picture (reinitialized, that is, limited to the most recent year alone) is a bit more appreciable, as seen in Figure 5.

Stopping for a moment, let us say that increasing risk is not surprising per se, but rather, the ICS confirms in a methodologically coherent way what might be your assumption, viz., that risk is rising in the aggregate but quite unevenly with respect to individual elements that make up the overall risk envelope.

Previous studies have shown that (non-nation-state) attackers manage their work lives in conventional ways—they have identifiable work shifts, they outsource where it is economical to do so, and they prefer tools that enhance labor productivity—all signs of normal work patterns. To those we might add seasonality. The visual indication of that seasonality is a plot, for each month, of whether a three-month sliding window of variability for each question increased from the previous month's value (counted and shown in blue) or declined (counted and shown in red). Figure 6 graphs Year 1 (of the ICS, April 2011 through March 2012), and Figure 7 graphs Year 2 (April 2012 through March 2013); they are rather similar.

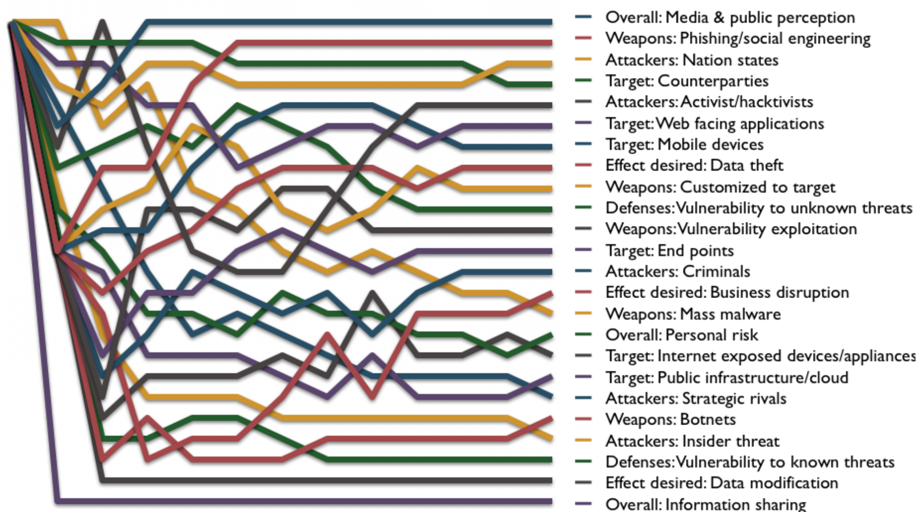


Figure 5: Tracking rank order just over the most recent year. As examples, phishing as a weapon increases while botnets decrease over the year.

R	Pair
0.942	Personal risk overall & Mobile devices as targets
0.928	Business disruption as the effect desired & Botnets as the weapon
0.924	Data theft as the effect desired & Phishing/social engineering as the weapon

Table 1: The amount of correlation for some pairs of risks

Despite the rat's nest in Figure 4 (the rank ordering of the month over month change in component risk contribution to the overall ICS), the trendlines of some component risks are indeed close to the trendline of the overall ICS. The three component risk trends most closely mirroring the overall ICS are, in order, attacks by nation states, attacks by strategic rivals, and the respondents' estimation of their own personal cybersecurity risk.

Let us repeat that we are observers here; the questions that are asked are all relative—relative to the individual respondent's own definitions ("what is malware?") and to the relation of this month's risk to last month's. Put differently, all the results of the ICS are ordinal scale, not interval nor ratio scale. It is our bias that the only purpose for any program of security metrics is decision support, and relative measures, meaning trendlines of ordinal values, are decision support personified. For planners and policy makers, the differentials between component risks, the volatility of component risks both interior to themselves and between each other, and the implications of presuming that this or that trend is long-term durable are all decision supporting.

From time to time, someone will publish a paper that purports to settle the hypothesis that cybersecurity failures affect the market capitalization of the affected firm. We ask, instead, whether there is some other indicator that represents the breadth of the financial market overall. One candidate is the Chicago Board Options Exchange's Market Volatility Index, known as VIX or, colloquially, the "Fear Index."

The short answer to whether there is any relation between broad financial market swings and similarly broad cybersecurity swings is "no, there is no general relation between VIX and the ICS," that is to say that cybersecurity risk, as described by those who deal with it most closely, and general market fear do not correlate. Nevertheless, you may be interested to note that the three component risks that most closely correlate with the VIX are, in order, attacks by hacktivists, attacks by strategic rivals, and public infrastructure/cloud as a target. Perhaps these three confirm, indirectly, where the public's attention does lie.

The three least correlated are all about vulnerabilities (exploitation attacks, failures of defense against known vulnerabilities, and failures of defense against unknown vulnerabilities), perhaps likewise confirming where the public's attention does not lie.

Because we are not modeling but rather observing, there is no requirement that the questions we have chosen be uncorrelated with each other, nor that they be mutually exclusive and collectively exhaustive. The component risks are indeed correlated with each other to a degree, and again sampling three pairwise correlations between questions, we find the results in Table 1, each of which is intuitive. That these (and other pairwise correlations) may be unsurprising should be thought of as confirming various decision support assumptions.

Besides the fixed set of questions asked each month, we also ask a special question. For this past September, the question was whether the respondent had discovered an attack aimed at some other party, and 65% had done so. This aligns with the report in the Verizon Data Breach Investigations Report [3] that 80% of data breaches are discovered not by the victim but by some unrelated third party. Another special question asked what percentage of the cybersecurity products now running in the respondent's own enterprise would the respondent reinstall if starting from scratch; we found 35% buyer's remorse (would not again buy or install). And another special question found that more than half of the respondents are unable to find qualified help regardless of the level of compensation that can be offered.

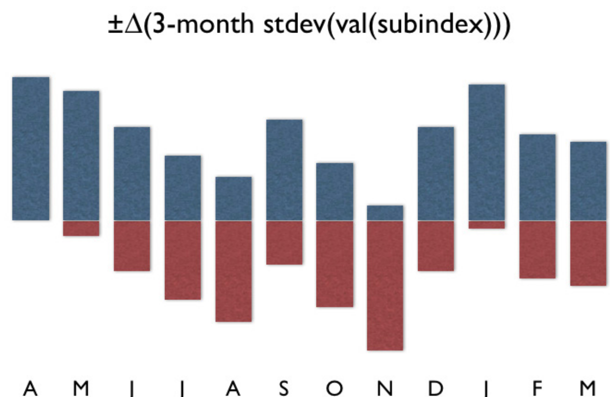


Figure 6: April 2011 through March 2012 showing increase (above the line) or decrease (below the line; red in PDF version)

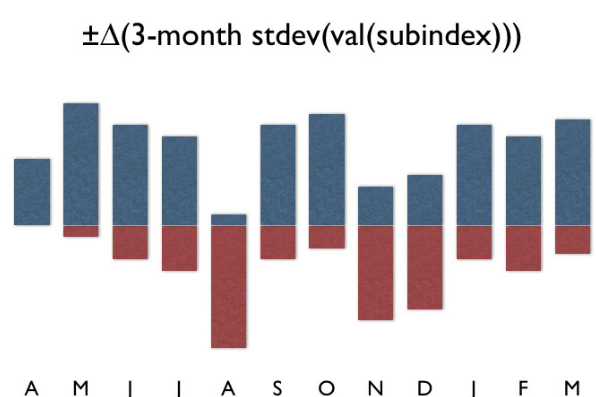


Figure 7: April 2012 through March 2013; looks a lot like Figure 6

You get the idea.

Because the greatest truth in cybersecurity practice is a robust rate of change, the ICS does occasionally have new questions added to the survey panel and old ones removed. The methodology for doing so mimics all established financial indices and is, by design, boring. In fact, everything about the design of the ICS seeks to be boring so that the results we obtain cannot be attributed to artifacts of methodology. Anyone familiar with survey research or index construction will find little to comment upon with regard to the ICS. All the juice is in the subject matter.

We are always interested in adding qualified individuals to the pool. We seek respondents who have direct operational responsibility for cybersecurity, and we seek them as individuals, not as representatives of their company's position on anything. We offer the respondent, in return, data not available to the general public. We do all our work with the ICS in a way that assures respondents of non-traceback of their participation and, more importantly, of their answers to the questions. (There are air gaps, data destruction, and other safeguards to that latter point about non-traceback that we will discuss one-on-one with candidate respondents.)

If you are, or can recommend, such a person, then please be in touch; the Web site says how to do this. Of course, suggestions are always welcome.

References

[1] www.cybersecurityindex.org.

[2] www.conference-board.org/data/consumerconfidence.cfm.

[3] www.verizonenterprise.com/DBIR.

/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

You may remember last month I outlined a failed job interview with Facebook. This month's curious encounter is with Amazon (the mega-corporation, not the river, although there are certain similarities).

I got a phone call at work one day from a cheery recruiter who wanted to discuss a job position with me. She then told me about the working conditions and benefits at Amazon for a good 15 minutes, at the close of which I told her I was sold. Then she asked me some really elementary questions about UNIX, such as, "what command would you use to see what processes are running." I told her much more than she wanted to know about ps, including an exposition on all the command-line options available depending on whether your kernel is derived primarily from AT&T or BSD. Then I talked about GNU ps on Linux. "Wow," she replied, helpfully.

"What's the job?" I finally asked, since that seemed a germane point. It was a systems engineer position. "That sounds like a lot of scripting," I said, warily. "Oh, it is," she enthused, "C, C++, Perl, Python, and Ruby, mostly." I rolled my eyes. "I've never been very good at any language that starts with C," I explained. "I was once decent at Perl. I fiddled around with Python when it first came out. I skirted the perimeter of Ruby, looking for an easy way in and found none. Java 1.0 and I had a short but intense relationship. Coding isn't my strong suit, in other words. How about a job involving information security?"

She sounded doubtful, "I don't think we have any of those, but I'll look."

"Amazon has no need for any security people? I find that hard to believe."

A minute or so later and she's back.

"I found one!"

"Great. What's the title?"

"Systems security engineer."

"Let me guess: it involves scripting with C, C++, Perl, Python, and Ruby."

"Yes, yes it does! How did you know?"

"Lucky guess. Look, all of my current certifications are in infosec. I haven't been certified as any sort of programmer since 1998. You should be able to tell that from my resume. Why did you call me?"

She hemmed and hawed for a moment. "Because AWS is getting into the classified information space and people with your security clearance are hard to find."

"Super. I'll tell you what: if you locate a bona-fide information security position, feel free to call me back. Otherwise, please take me off your list of people whose rusty or non-existent

skills we're willing to overlook just because they won't have to sit around on their thumbs for eighteen months to get the right clearance."

As you can probably surmise from my "job recruitment disaster chronicles," I've experienced a sudden surge in activity from headhunters lately. From my observations this is always a good sign for the economy, if nothing else. In the course of listening to/reading their spiels, I had a Scythian moment (a public mention for the first person who gets this reference and communicates with me) and got fixated on the language they were using to describe UNIX/Linux sysadmin positions instead of comprehending the totality of the recruitment effort itself. This led, inexorably, to scouring the InterWebs in a quest for the perfect representative job announcement. Below you will discover—verbatim, spelling and grammatical errors intact—the fruits of this labor. Don't forget to scoop out the pits first or you'll break your teeth.

Position ID: aX075ENGir4zY1eJe4bk1X

The UNIX System Administrator must possess a high level knowledge of the UNIX/Linux OS. Must be highly proficiency at the UNIX command line involving a variety of utilities, interpreters and compilers and have Shell Programming skills. In addition, high proficiency in Operating Systems. Works on unusually complex technical problems and provides solutions that are highly innovative and ingenious. Establish a defined framework as an analytic environment. Experience with System Management tools to include NetApp Management, and tools such as SPLUNK, NAGIOS, ZENOSS, GANGLIA, or CLOUDERA. Experience or ability to perform HADOOP System Administration, installation, configuration, networking, server administration/management, troubleshooting, security, monitoring, tuning, capacity planning, backup/recovery, service pack/hot fix/patch install, vendor support liaison, coordination with technical staff at all levels.

Experience with NoSQL technologies (Cassandra, HBase), NoSQL Data Warehouse technologies (HIVE), Lucene, Tomcat, Pentaho, OpenLDAP, BIND/DNS, YUM, Puppet, Cacti, VMware, KVM, Xen, Hypervisor, OpenVZ, Brocade Fiber Channel experience in zoning, Memcache, Redis, experience using Chef to deploy and manage large clusters of servers, AllFusion ERwin Data Modeler; Data modeling software; IBM Rational Data Architect; Visual Paradigm DB Visual ARCHITECT, Gluster FS, Drupal, Sendmail, Postfix, Cyrus, Dell Open Manage, IT Assistant, What Up Professional, Spine, Openstack, Spacewalk/RedHat Satellite, F5, HP Lefthand, EMC, Elastic Search, Voldemort, NFS, DAS, NAS, and SAN. Assures Data Center is clean and clear of obstructions.

Ideal candidate will be a bright, enthusiastic, flexible, energetic, and results-driven professional who works best in a multi skilled team, sometimes during non-core business hours. The company has a very casual and great culture. They house an indoor gym with accessible classes and showers, state of the art kitchen on site and promote a 37.5 hour work weeks along with many other perks! They are very focused on "work-life" balance. This is the type of company that you will want to grow with long term. Reasonable regular, predictable attendance is essential. Requires hands and fingers dexterity to operate computer components such as a keyboard or mouse. Must possess sufficient peripheral vision to have the ability to observe an area that can be seen up and down or the left and right while eyes are fixed on a given point. Ability to identify and distinguish colors essential. To perform this job successfully, an individual must be able to perform each essential duty satisfactorily. Must have knowledge of the structure and content of the English language including the meaning and spelling of words, rules of composition, and grammar. Must possess the ability to combine pieces of information to form general rules or conclusions (includes finding a relationship among seemingly unrelated events). Must have the ability to arrange things or actions in a certain order or pattern according to a specific rule or set of rules (e.g., patterns of numbers, letters, words, pictures, mathematical operations) and the ability to generate or use different sets of rules for combining or grouping things in different ways.

Perks include: Weekly FreshDirect grocery delivery & weekly catered lunch. Classic Pac-Man/Galaga arcade machine. Coke machine stocked with free drinks.

I'm off to update my resume now. I think I remember a little GW-Basic. GOTO rules!

BOOKS

Book Reviews

ELIZABETH ZWICKY, MARK LAMOURINE, AND MELISSA GRAY

Peopleware

Tom DeMarco and Timothy Lister
Addison Wesley, 2013. 238 pp.
ISBN 978-0-321-93411-6

Reviewed by Elizabeth Zwicky

Peopleware is an old favorite of mine, and I approached this edition with some trepidation, the way you approach anything you loved when younger that has now been updated; will it turn out to have lost its luster, either through age or through savage updating? On the whole, I was very happy. The original copyright shown is 1987, and in the intervening roughly quarter-century, a lot has changed, but the fundamentals of programming and managing people have not. The update manages to remove most of the dated references and adds a good bit of purely new material.

Peopleware is an introduction to the human side of managing technology teams. It is eminently readable—it comes in short, vivid chunks that say things programmers want to hear in terms that management can understand. If you are feeling that there is something fundamentally missing from the practice of technology management, this will fill that gap and fire you up.

I'm somewhat sad that after this long, the humanistic approach found here still feels fresh, startling, and avant-garde. Paying more attention to human issues than new technologies, like personal jet packs and hover cars, seems destined to remain the wave of the future. And yet there are signs of hope—when a phone rings audibly in my office, people are startled and displeased. It's a rare event, rare enough that the last time a repetitive noise went on for a while, one of my colleagues leapt up angrily to search out and silence the phone, only to realize that the rest of the office was laughing at him. The annoying noise was in fact a crow on the windowsill. The good news here is that our office environment is both quiet and near a window; the bad news is that the entire team was within sight and earshot of the crow and the ensuing search. So there's still work for *Peopleware* to do.

Adaptive Software Development

James A. Highsmith
Addison Wesley, 2000. 348 pp.
ISBN-0-932633-40-4

Reviewed by Elizabeth Zwicky

Somehow this crept onto a list of new releases, so I was puzzled to read through an entire book on software development practices for rapidly changing environments that never used the terms "Agile" or "Extreme" as we now know them. It's still a worthwhile book, with a detailed explanation of a practical and human-centered approach to development in high-change environments. It is quite kind to the waterfall model, suggesting places it is appropriate and ways to gently move people away from it. And it is heavily influenced by *Peopleware* while being much more traditional in tone and format.

This would be a great bridge book for somebody who wants or needs to move to a more flexible style of managing projects, but would like to do so without overt, radical breaks with tradition.

The Practice of Network Security Monitoring

Richard Bejtlich
No Starch Press, 2013. 334 pp.
ISBN 978-1-59327-509-9

Reviewed by Elizabeth Zwicky

This book will tell you how to install Security Onion and its add-ons, how to work with those tools (including tricks, traps, and subtleties involved), and it provides significant discussion of how to place monitoring taps. Bejtlich provides some advice on how you keep track of what's going on when you don't know what you're dealing with. These are all significant challenges for new network security administrators.

I feel convinced that this book would help me set up a network security monitoring system based on open source systems and use it to improve the security of pretty much any network. On the other hand, this is a task where I don't really need all that much help—I know a lot about how networks work and about the practicalities of securing them. I'm less convinced that somebody without all that background would find it sufficient.

What it doesn't talk about, except in vague and abstract terms, is the actual practice of network security monitoring—what alerts are important? which ones are not? There's a lot of good information here, but it doesn't quite jell into a clear problem statement and answer, and it isn't quite enough for a security novice.

Graph Databases

Ian Robinson, Jim Webber, and Emil Eifrem
O'Reilly Media, 2013. 200 pp.
ISBN 978-1-449-35626-2
Reviewed by Elizabeth Zwicky

In a world where we all used graph databases, you could just write a book about how to use them. But most of us don't, so this is a relatively broad introduction, covering what graph databases are, why you might want one, and how you would design, query, and optimize one.

Simplifying somewhat further than is advisable, a graph database allows you to query data by talking about objects and relationships instead of by talking about rows and fields. Graph databases are considerably more efficient at certain kinds of queries than traditional databases. If you want to ask "What did Customer 43 order recently?" any old database will do. If you want to ask "What might Customer 43 order next?" you will rapidly find yourself asking, "What other customers ordered the same items as Customer 43?" and a suitably designed graph database will vastly improve your experience. (Or so the authors claim, very believably.)

If you're in a position to implement a system using new database technology, graph databases are an interesting tool to have available to you, and this introduction, while it clearly doesn't cover all the corners of the space, should get you started.

Advanced Programming in the UNIX Environment, 3rd Edition

W. Richard Stevens and Stephen A. Rago
Addison Wesley, 2013. 994 pp.
ISBN 978-0-321-63773-4
Reviewed by Mark Lamourine

A lot has changed since I first read the late Richard Stevens' *Advanced Programming in the Unix Environment*. Stephen Rago has just released his second update. There are very few books I enjoy rereading and fewer still tech books, but I enjoyed this refresher course.

Advanced Programming describes the interface between programs and the kernel in a *NIX system. Even when a program uses higher level libraries, in the end this is what they come down to.

There are numerous tutorials on programming languages and programming in general. There are textbooks describing the *NIX kernel internals. Stevens and Rago don't just list the kernel system calls and their arguments. They illustrate their use and the behavior of the kernel in response. This gives the reader a sense not just of how to use each call, but when and why. It also gives them the ability to work backward from the behaviors of a system to the calls that would be the cause.

In 1993 the systems described were AT&T System V R4 and 4.3BSD. In 2013 Rago has added FreeBSD, Linux, MacOS, and Solaris 10 (arguing that while Solaris is derived from SYSVR4, it has 15 years of enhancements). While this might seem to add quite a bit, it seems that standardization has largely served its purpose. Variations still exist but they're not nearly as large as might be expected. The 3rd edition is almost 1,000 pages compared to 740 for the first edition, but Rago has added two sections on threading and one on network sockets to address topics that didn't exist in the early 1990s.

There are remarkably few actual system calls (functions that cause a process to switch from user to kernel mode). The remainder of the interfaces are known as system libraries and are generally built on top of the system calls. These are used to manage the core system resources (files, processes, threads, and memory) to communicate between processes (signals, semaphores, shared memory) and between systems and devices (serial I/O and networking). Stevens and Rago explore each of these in some depth, highlighting differences between operating system flavors.

The authors begin most chapters by explaining some aspect of a running *NIX system: files and I/O, processes and interprocess communication, errors and the process environment variables. They show what each feature is for, how it affects the operation of the system, and how programs interact with it. Only then do they introduce the system calls with realistic example code. Each chapter closes with a traditional summary and a set of exercise questions.

Stevens and Rago close the book with a couple of chapters that present complete uncontrived examples of real-world systems programming.

Advanced Programming is known as a classic for good reasons. The writing is clear and precise. The examples are detailed but to the point. The chapters follow a progression from topics that will likely be familiar and commonly used to those that may be more specialized or esoteric. The rationale for or history behind a design choice or variation is provided when it offers some insight into how a feature is to be used. This is one of the rare books that works both as an introduction and as a reference.

I've done a fair amount of systems level programming and I recommend *Advanced Programming* to pretty much anyone who programs *NIX systems seriously; however, I'm a system administrator by trade and avocation. There are two divided and vocal camps on the question of whether programming is required for system administration. I won't weigh in on the question of requirement, but I don't think it can hurt to at least learn how to

read C code. I think it can be a tremendous benefit to understand the kernel system calls, especially when tracing and debugging processes. I recommend *Advanced Programming* to anyone who's interested in understanding the interfaces between *NIX programs and the system that runs them.

The Go Programming Language Phrasebook

David Chisnall
Addison Wesley, 2012. 264 pp.
ISBN 978-0-321-81714-3

Reviewed by Mark Lamourine

In *The Go Programming Language Phrasebook* David Chisnall provides all of the information an experienced coder needs to begin experimenting with Go. He doesn't spend a lot of time on the minutiae of the language and libraries, deferring instead to other books and resources when a reader might want more detail. He concentrates on the features that make Go significant and on the idioms and coding patterns that make the best use of those features.

It turns out that Go is designed not to illustrate some new programming paradigm, but in response to the known shortcomings of the aging C programming language in the context of system software development where it still dominates. Much of the Go syntax looks like C, but where it differs there is a reason. Usually the changes are meant to eliminate common coding errors or to decrease the complexity of implementing modern coding patterns. The most significant new features, "goroutines" and "channels," provide a cleaner means of implementing concurrency both on individual multicore computers and in networked distributed systems. It is also notable that, although Go is a compiled language, the development environment offers a way to run many programs from source code on the command line as if they were scripted.

The author avoids the worst impulses of writers of this kind of book. The phrasebook format can lead an author to provide a code snippet for every variation of every feature of every library. Chisnall focuses on writing about Go and uses the code snippets only to illustrate a point. He details not just how Go differs from other common languages but *why*. Because Go is meant to replace C, a low-level language, the machine details, such as the placement of structures in memory, will peek up through to the coder. Chisnall doesn't shy away from discussing how coding style and idiom will affect the behavior of the machine and how Go features contrast with other languages. Go is still a young language, and Chisnall informs the reader where there are caveats, gaps, or areas of continuing development that might make his examples obsolete.

In addition to the standard language primer and features (variables and types, scoping, objects, arrays, and collections) and the new features (goroutines and channels), Chisnall includes sections on working with the Go runtime environment, packaging and distributing code, and debugging. The one thing notably missing is any mention of a unit testing framework.

The Go Programming Language Phrasebook is an excellent introduction both to a new alternative for systems programming and a survey of the challenges faced by coders implementing modern concurrent and distributed applications. Because Go produces executable binaries for any modern OS and architecture, I will certainly consider trying it the next time I need to code a binary from scratch, and this book will be the first source I pick up.

Realm of Racket: Learn to Program One Game at a Time!

Matthias Felleison, David Van Horn, Conrad Barski, Forrest Bice, Rose DeMaio, Spencer Florence, Feng-Yun Mimi Lin, Scott Lindeman, Nicole Nussbaum, Eric Paterson, Ryan Plessner
No Starch Press, 2013. 294 pp.
ISBN 978-1-59327-491-7

Review by Mark Lamourine and Melissa Gray

I've read a number of books aimed at introducing software development to new readers, but I wouldn't have picked Lisp as a first language. *Realm of Racket* is an introductory text aimed at college freshman and written at least in part by students at Northeastern University. The students get top billing on the cover. The language is Racket, a derivative of Scheme, which is in turn a Lisp variant. The authors try to set an informal tone with comic strip artwork and a game and quest narrative which seemed to me to be a bit childish for the audience. It's been a long time since I was a college freshman.

Luckily I had a handy intern in the cube across from me, and she agreed to read it and give me her impressions. On reading them I had to reconsider my first take on the book. This is what she had to say:

The information is laid out in an accessible and engaging way. I think it would be effective and understandable for college freshmen regardless of previous programming experience. The story and cartoons are engaging. High-level material is clearly explained and given to the reader gradually in a way that builds on the previous chapters. As someone who has taken both high school and college intro programming courses, game-based examples and exercises are a good way to teach logic and decision-based programming. So I think this is a strong feature of this book.

The "read front to back" method this book employs might be bothersome for impatient students who would rather skip through a textbook by topic. However, if a student's goal is really to learn the material, this ends up being a great method because you can learn things in a logical order.

So, Melissa didn't seem to be as put off as I thought she'd be. When I asked her about it she shrugged and directed me back to the text and the teaching arch and I had to take a new look.

The chapter topics and sequence presented are not what I've come to expect for procedural languages, but are natural for Lisp. Variables, conditionals, and functions come first, but then the authors present recursion and lambdas before coming back to looping constructs and trees. They don't stop there, though, and this is where the youthful comic strip cover seems misleading. The authors continue, introducing more advanced topics, memoization, and lazy evaluation. The book closes with several chapters developing a simple distributed game using client-server constructs and messaging.

There's a lot packed into this book and it's not really aimed at the tweens that some other No Starch programming books have been, though I wouldn't hesitate to offer it to a motivated high school student. The DrRacket IDE runs on Windows, MacOS, and Linux so that students can begin work in whatever environment they are comfortable. The IDE is also fairly comprehensive, containing tools for interaction, development, and debugging. The Racket language includes module constructs that I don't remember seeing when I learned Scheme. DrRacket also provides a GUI library that I know wouldn't work on my VT100.

In the end I'm impressed. *Realm of Racket* and DrRacket both are well thought out and well suited to their tasks.

USENIX Board of Directors

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

PRESIDENT
Margo Seltzer, *Harvard University*
margo@usenix.org

VICE PRESIDENT
John Arrasjid, *VMware*
johna@usenix.org

SECRETARY
Carolyn Rowland
carolyn@usenix.org

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

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

Sasha Fedorova, *Simon Fraser University*
sasha@usenix.org

Niels Provos, *Google*
niels@usenix.org

Dan Wallach, *Rice University*
dwallach@usenix.org

CO-EXECUTIVE DIRECTORS
Anne Dickison
anne@usenix.org

Casey Henderson
casey@usenix.org

Nominating Committee for USENIX Board of Directors

The biennial election of the USENIX Board of Directors will be held in early 2014. The USENIX Board has appointed Margo Seltzer to serve as chair of the Nominating Committee. The composition of this committee and instructions on how to nominate individuals will be sent to USENIX members electronically and will be published on the USENIX Web site this fall.

USENIX ASSOCIATION FINANCIAL STATEMENTS FOR 2012

The following information is provided as the annual report of the USENIX Association's finances. The accompanying statements have been reviewed by Michelle Suski, CPA, in accordance with Statements on Standards for Accounting and Review Services issued by the American Institute of Certified Public Accountants. The 2012 financial statements were also audited by McSweeney & Associates, CPAs.

Accompanying the statements are charts that illustrate the breakdown of the following: operating expenses, program expenses, and general and administrative expenses. The operating expenses for the Association consist of the following: program expenses, management and general expenses, and fundraising expenses, as illustrated in Chart 1. The operating expenses include the general and administrative expenses allocated across the Association's activities. Chart 2 shows the breakdown of USENIX's general and administrative expenses. The program expenses, which are a subset of the operating expenses, consist of conferences and workshops, programs (including *;login:* magazine) and membership, student programs and good works projects, and the LISA Special Interest Group; their individual portions are illustrated in Chart 3.

The Association's complete financial statements for the fiscal year ended December 31, 2012, are available on request.

—Anne Dickison, Co-Executive Director

—Casey Henderson, Co-Executive Director

USENIX ASSOCIATION
STATEMENTS OF FINANCIAL POSITION
As of December 31, 2012 & 2011

ASSETS	2012	2011
Current Assets		
Cash & cash equivalents	$ 552,100	$ 787,118
Receivables	85,858	56,975
Prepaid expenses	74,531	53,320
Total current assets	712,489	897,413
Investments at fair market value	5,118,785	5,177,383
Property and Equipment		
Office furniture and equipment	364,776	351,131
Website	640,713	345,394
Leasehold improvements	29,631	29,631
Less: accumulated depreciation	(445,007)	(330,721)
Net property and equipment	590,113	395,435
Other assets	46,961	337,960
	$ 6,468,348	$ 6,808,191
LIABILITIES AND NET ASSETS		
Current Liabilities		
Accounts payable	$ 490,299	$ 543,430
Accrued expenses	95,554	46,250
Deferred revenue	35,250	84,435
Total current liabilities	621,103	674,115
Long-Term Liabilities	46,961	337,960
Total liabilities	668,064	1,012,075
Net Assets		
Unrestricted Net Assets	5,800,284	5,796,116
Net Assets	5,800,284	5,796,116
	$ 6,468,348	$ 6,808,191

USENIX ASSOCIATION
STATEMENTS OF ACTIVITIES
For the Years Ended December 31, 2012 & 2011

	2012	2011
REVENUES		
Conference & workshop revenue	$ 3,510,227	$ 3,033,125
Membership dues	326,834	364,450
Product sales	975	902
LISA SIG dues & other revenue	65,759	96,863
General sponsorship & ads	3,005	5,000
Total revenues	3,906,800	3,500,340
OPERATING EXPENSES		
Conferences & workshops	3,018,520	2,604,390
Membership ;login:	533,356	431,357
Projects & good works	44,889	161,180
LISA SIG expenses	82,367	150,115
Management and general	607,115	466,559
Fund raising	49,622	57,503
Total operating expenses	4,335,869	3,871,104
Net operating deficit	(429,069)	(370,764)
NON-OPERATING ACTIVITY		
Interest & dividend income	143,291	172,953
Gains (losses) on marketable securities	356,587	(7,448)
Loss on disposition of equipment	(3,987)	-
Investment fees	(62,341)	(63,285)
Other non-operating	(313)	(2,702)
Net investment income & non-operating expense	433,237	99,518
Change in net assets	4,168	(271,246)
Net assets, beginning of year	5,796,116	6,067,362
Net assets, end of year	$ 5,800,284	$ 5,796,116

Chart 1: USENIX 2012 Operating Expenses

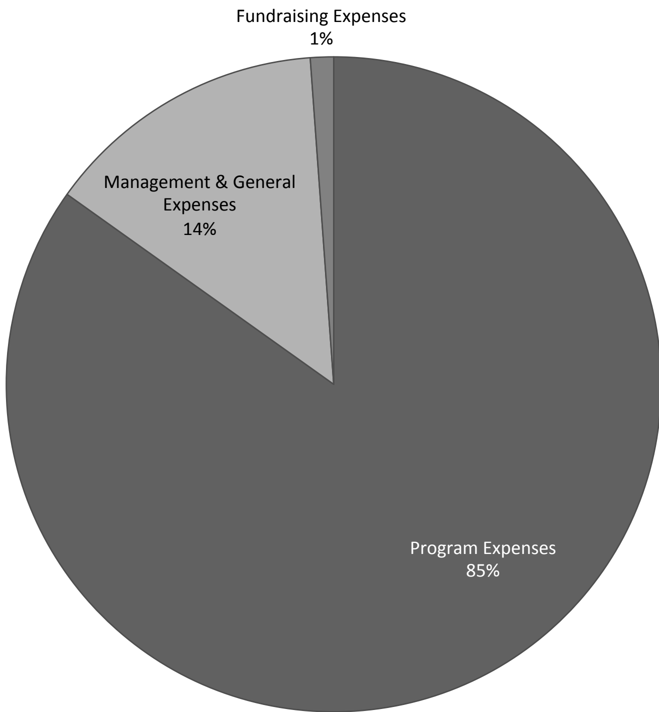


Chart 2: USENIX 2012 General & Administrative Expenses

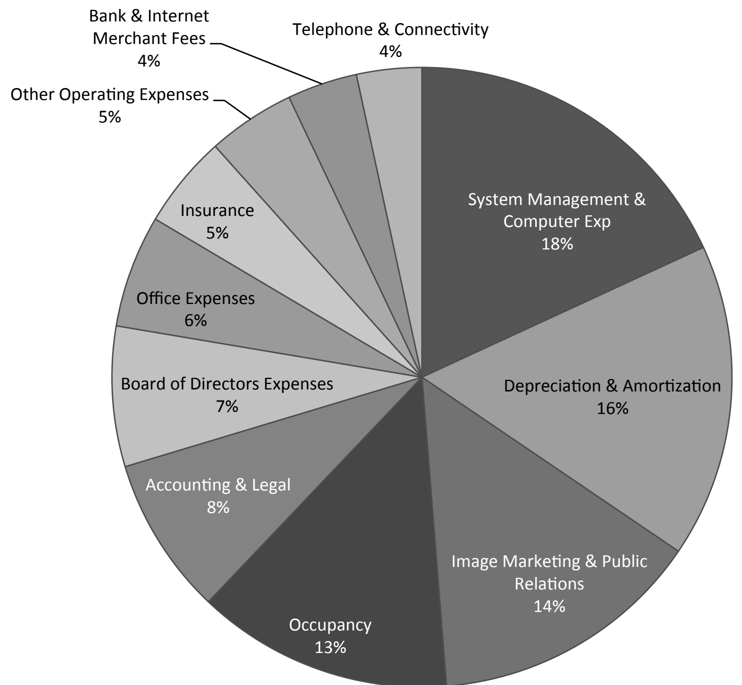
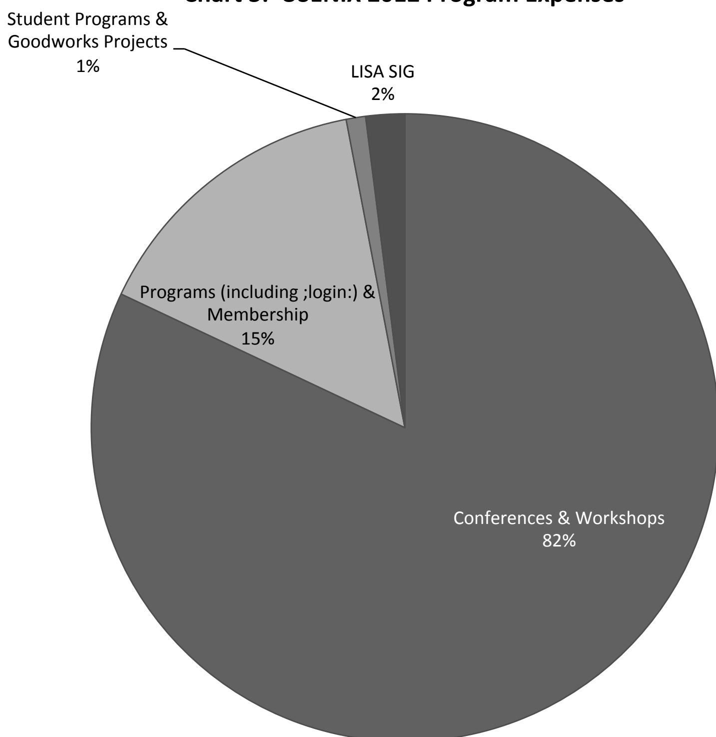


Chart 3: USENIX 2012 Program Expenses



REPORTS

Conference Reports

In this issue:

HotOS XIV: 14th Workshop on Hot Topics in Operating Systems 80
Summarized by Rik Farrow, Seungyeop Han, William Jannen, Shriram Rajagopalan, Deian Stefan, Jonas Wagner, Edward Yang, and Cristian Zamfir

HotPar '13: 5th USENIX Workshop on Hot Topics in Parallelism 95
Summarized by Rik Farrow

The complete reports from HotPar '13, USENIX ATC '13, ICAC '13, HotCloud '13, HotStorage '13, and WiAC '13 are available online at www.usenix.org/publications/login

HotOS XIV: 14th Workshop on Hot Topics in Operating Systems

Santa Ana Pueblo, NM
May 13-15, 2013

HotOS XIV Opening Remarks

Summarized by Rik Farrow (rik@usenix.org)

Petros Maniatis, Intel Labs, the PC chair, explained the ground rules for the HotOS '13. Presenters had only 10 minutes, with a few minutes for questions and answers as the next presenter set up his or her laptop. Each talk session was followed by a half-hour open mike session, where participants were welcome to speak on any topic, although the discussions were generally related to ideas brought up during the previous session or earlier in the workshop.

Petros also introduced a new concept: unconference sessions. Four sessions were set aside for groups to meet about topics of their own choosing. Attendees announced topics during a session on Monday morning and gave reports on the issues, and sometimes on the results of these meetings, on Wednesday, right before the end of the workshop.

Shuffling I/O Up and Down the Stack

Summarized by Shriram Rajagopalan (rshriram@cs.ubc.ca)

We Need to Talk About NICs

Pravin Shinde, Antoine Kaufmann, Timothy Roscoe, and Stefan Kaestle, Systems Group, ETH Zurich

Timothy Roscoe began by pointing out that modern NICs have become complex devices with a variegated set of features, but operating systems do not provide proper abstractions to access many of these features. Windows provides different abstractions for each NIC manufacturer, whereas Linux does not provide any support to access the hardware functionalities in modern NICs. Most operating systems as of now cannot optimize performance of a workload by automatically identifying and leveraging functionalities exposed by the NIC hardware.

Dragonet presents a new network stack design that represents the protocol state machine in the OS as a dataflow graph. The NIC's capabilities are represented as a dataflow graph as well. The two graphs can be combined in such a way that functionalities not provided by the NIC hardware can be provided by software components in the network stack.

Someone pointed out that graphics folks have taken a similar approach, and asked whether Mothy could draw a parallel between the two approaches. Mothy replied that their approach has a similar flavor; however, graphics cards are heterogeneous and provide arbitrary multiprocessing capabilities apart from functionality offload. His team is dealing with fixed function hardware. Someone else asked how high should the abstractions go up the stack: for example, the ability to push computations onto the NICs for certain workloads (e.g., receiver side scaling). Mothy answered that they don't know yet, but that they'd like to be able to offload processing to the NIC, but they need to track the spatial placement of threads. Brad Karp (University College, London) asked whether it is possible to automatically capture the NIC's capabilities in a protocol graph, when its firmware is updated, and if so wouldn't they have to update the OS's protocol graph accordingly. Mothy responded that you could treat this issue like a bug fix for bad firmware in the card. Until the firmware is fixed, the OS could use a different resource graph as a workaround. Their design just makes it easy to work around these hardware issues.

The NIC Is the Hypervisor: Bare-Metal Guests in IaaS Clouds

Jeffrey C. Mogul, Jayaram Mudigonda, Jose Renato Santos, and Yoshio Turner, HP Labs

Jeff Mogul started with a question: Why would anyone want to run a bare metal guest without a hypervisor? There could be several motivations, such as performance, security, application/vendor support for certain software, licensing requirements, and customer demand. The next question that naturally arises is how can one run both bare metal guests (BMGs) and virtual machines in the same cloud? With BMGs, we no longer have a guest OS running over a hypervisor, so where will the protection boundary be drawn? Jeff suggested using the Switch/NIC to enforce a hypervisor-like protection boundary for BMGs.

A simple inventory shows that we have several components already in place. For example, a sNICh provides ACLs with hardware NICs. Remote management can be accomplished via components such as HP's iLO (or equivalents from other

vendors, using IPMI) with little modification. The Remote Management Engine (RME) at the end host interacts with the cloud controller. Depending on the requirements of the BMG, the RME configures the NIC with appropriate protection boundaries by disabling certain features; however, other things, such as checkpointing, migration, etc., require guest OS support. Jeff suggested that using an SDN is not the appropriate solution because BMG-NICs present a cleaner separation between the edge hardware and the network fabric and scales better.

Someone asked whether customers who demand bare-metal guests have concerns with licensing fees. Jeff answered that some applications cannot run on a VM, and apps would not be able to tell they were running over a sNICh. Muli Ben-Yehuda (Technion) asked whether this would still be necessary if the hypervisor had no performance penalty. Jeff pointed out that performance is just one aspect. A key driving factor for BMG-NICs is licensing and support requirements. Someone asked about the problem with RMEs accessing the main memory, and Jeff replied that because of their design (the BMC interface used by IPMI) RMEs do not have a main memory map. Another person asked why the RME is even relevant. Jeff said that they need someone to control the NIC. Current systems allow RME to control the NIC. Basically, we are leveraging something that's readily available.

Virtualize Storage, Not Disks

William Jannen, Chia-che Tsai, and Donald E. Porter, Stony Brook University

Bill Jannen stated that virtualization works great because of hardware emulation but has a big performance impact on storage. For example, we have duplicated storage stacks in both the guest and the host—things such as page caches, read ahead blocks, etc.—when using a file-based backing disk. The double caching can cause correctness problems with certain file system operations in the event of failure. Bill described an example scenario where the guest issues an unlink system call on a file and gets an acknowledgement from the host; however, at the host level, the inode information still resides in the page-cache. Should the host fail and come back up, the guest OS's application would see the deleted file and might react in an undefined manner.

They proposed separating the media access layer from the file system. The application interfaces would reside in the guest while things like I/O schedulers would be at the host. They could then augment the guest API with performance, ordering hints, etc.

Steve Niel (VMware) claimed that VMware ESX servers do not have this issue; however, he appreciated the idea that we need to modularize the storage layer. Ed Yang (Stanford) said that this also applies to Xen and KVM, and that their example pertains to the configuration settings for their guest OS. Muli Ben-Yehuda said that the idea of modularizing certain aspects of storage, such as file systems, depends totally on the data structures that the file system uses. The case may be that such modularization is not possible for a given file system due to the nature of its data structures.

Unified High-Performance I/O: One Stack to Rule Them All

Animesh Trivedi, Patrick Stuedi, Bernard Metzler, and Roman Pletka, IBM Research Zurich; Blake G. Fitch, IBM Research; Thomas R. Gross, ETH Zurich

Animesh Trivedi stated that I/O performance has changed over the years. We have moved from disks to flash and will move to PCM, which represents two to five orders of magnitude performance improvement; however, the OS is not leveraging these features. We need a set of rich I/O semantics with direct access to hardware.

High performance I/O stacks work great with disks but don't perform well with NVRAMs. Instead of reinventing the wheel, he suggested, let's leverage the technology available in the networking community. Inspired by high performance software-controlled NICs, he proposed user-space mapped I/O channels with no OS involvement. An even better alternative would be to unify both I/O stacks. The OS could support a single set of abstractions for multiple sets of devices. The application would no longer care whether the storage is local or remote. Animesh said they have a working prototype that performs two to five times better with about a half million IOPS.

Muli Ben-Yehuda disagreed with Animesh's claim that network performance issues with respect to application access have been fully solved. Animesh replied that they do not claim that it's fully solved. Their opinion is that certain aspects of this space have been fully fleshed out and they propose to leverage them. For example, the OS would do a one-time translation to set up the I/O channel, acting like a control plane, for a very large file transfer. John Ousterhout (Stanford) asked what if there were a very large number of small files, which would be doing too many checks and hurting latency. Animesh agreed that too many data/control plane switches would have an impact on performance. Ed Bugnion (EPFL) pointed out that in networks, the socket is the central abstraction. In storage, its equivalent is SCSI. Their example is to use a niche network example (direct hardware access) and build a system on top of it. So at best, it's a niche within a niche. Animesh countered that sockets don't do high-speed transfers of hundreds of GBs of data. If you need high performance I/O, you need a niche. Simon Peter (U Washington) asked, what if two applications want to access the same file? Animesh said that you just remap the same channels with multiple processors and assume that the hardware can keep track of the ordering. Andrew Warfield pointed out that Animesh had focused on the similarities between the two domains, and asked that Animesh provide a big difference that is challenging. Animesh replied that networks have no notion of transactions while storage uses a lot of transactions. We have no way to roll back a transaction when doing I/O over network (but we can over storage).

Open Mike

Matt Welsh (Google) asked whether we know the kind of applications that are driving the kinds of papers that were seen in the I/O session. Do all applications need these features, such as direct access to I/O, or is it just a few? Timothy Roscoe responded that trading applications is a good use case because they cannot afford the hit on latency. He agreed that the customer base was a small one and that the application domain for these ideas was small.

Jeff Mogul commented that HPC applications are difficult to manage as they grow—especially resources, I/O, etc. The concepts presented in the session basically proposed abstractions that help the application/user easily manage these resources. Alex Snoeren (UC San Diego) added that, although these papers proposed to take the hardware capabilities to user space, hardware vendors (e.g., storage) are moving in the other direction (keeping to kernel space) in an effort to be compatible with each other. They don't want user-space libraries directly accessing their devices and creating compatibility issues.

Muli Ben-Yehuda reiterated Alex's observation that vendors are trying to move interfaces to the kernel because of legacy applications. He added that a major issue with direct hardware access is the loss of ability to migrate VMs and cited SR-IOV as one example. For enterprises with legacy applications, migration is a valuable tool compared to direct hardware access. Dave Ackley (U New Mexico) pointed out that NICs are getting smarter; it's the manifest destiny of silicon. Just as GPUs have been growing in capabilities by leaps and bounds, expect the same thing to happen with network processing. George Candea (EPFL) wondered whether a coordinated hardware/software design is needed to get the desired performance. The current approach is a real hodgepodge. Andrew Warfield (UBC) said that the current network stack is a real mess, with 15 vendors and only two of them focused on performance. Muli reiterated that moving code into user space wouldn't work for legacy applications. Steve Hand (Cambridge and MSR) said that once you bypass the hypervisor, you can no longer migrate, and people like the ability to do migration. So is this what customers really want?

Petros summarized by saying that this is a puzzle with multiple sides. Being able to mix-and-match and optimize for a particular solution would be nice. All sides have a point here—splitting things into small pieces, pushing some into hardware.

Edgy at the Edge

Summarized by Jonas Wagner (jonas.wagner@epfl.ch)

The Case for Onloading Continuous High-Datarate Perception to the Phone

Seungyeop Han, University of Washington; Matthai Philipose, Microsoft Research

Seungyeop Han introduced the case for onloading continuous high-data-rate perception onto the phone by explaining how computer vision has reached maturity and enables many applications, from context-sensitive reminders to tracking the user's diet. To perform sensing on the phone for continuous availability, cost, and privacy is desirable. Trends in memory size, processor speed, and power consumption indicate that this will be feasible in 2015.

A key optimization for on-phone video processing is using other sensors to gate the computation. These sensors identify frames that need not be processed, e.g., due to low light or motion blur, and discard more than 98% of all frames. This gating framework, combined with privacy concerns and the possibility to share models and algorithms between apps, calls for implementing video processing as an operating systems service.

Vova Kuznetsov (EPFL) asked whether gating is still useful if interesting frames come in batches. For many applications, gating still provides considerable energy savings. Matt Welsh (Google) asked whether this is really an OS problem. Seungyeop replied that techniques such as gating require multiple resources to be scheduled and shared between apps. Also, the OS can ensure privacy in the presence of malicious apps. To a follow-up question on privacy, Seungyeop replied that there are further ideas: for example, filtering an audio frame such that it is possible to identify the speaker but not the content. When asked whether his work makes offloading obsolete, Seungyeop said that, although some classes of applications require the cloud for reasons like low latency, more effort should go into onloading perception onto the phone.

Making Every Bit Count in Wide-Area Analytics

Ariel Rabkin, Matvey Arye, Siddhartha Sen, Vivek Pai, and Michael J. Freedman, Princeton University

Wide-area analytics need to cope with huge data volumes that exceed and outgrow the available bandwidth. Because not all data can be transmitted to a central location for analysis, existing systems make static decisions about what data to collect. They incur high costs for collecting (too) much data, yet are unable to obtain more data retroactively if the need arises.

Ariel Rabkin presented an alternative architecture in which full data is stored close to where it is collected. The data is then aggregated, summarized, and transmitted to the user with a precision and granularity that meets bandwidth constraints. The architecture supports reasoning about the bandwidth requirements of queries. Users can interactively define a policy that controls how results degrade gracefully as bandwidth changes. The OLAP cube is the chosen data model, because it supports merging, summarizing, and aggregating data automatically according to this policy.

When Doug Terry (MSR) asked about other data models that have been considered, Ariel replied that they had looked at SQL tables and MapReduce tuples. SQL tables require too much semantic awareness, especially in the presence of missing data.

Alex Snoeren (UCSD) recalled a similar, more general system where custom merge procedures could be specified for every data element. Ariel replied that such merge procedures are difficult to write for rich data, and hard to optimize compared to OLAP cubes. Peter Bailis (UC Berkeley) asked how the system compares to the Tiny Aggregation Service (TAG) used in sensor networks. Ariel explained that the focus is less on reliability and more on using the bottlenecked wide-area link as efficiently as possible.

QuarkOS: Pushing the Operating Limits of Micro-Powered Sensors

Pengyu Zhang, Deepak Ganesan, and Boyan Lu, University of Massachusetts Amherst

Pengyu Zhang presented work that pushes the operating limits of tiny sensors, such as medical implants or self-powered cameras. These harvest energy from temperature gradients, electromagnetic waves, or ambient light to charge energy buffers with a capacity of only few µAh. This severely restricts the amount of work that can be done in a single charge-discharge cycle, and precludes the use of conventional sensor-network operating systems.

QuarkOS fragments tasks as much as possible so that individual fragments stay within the energy limits. QuarkOS efficiently measures available energy and inserts sleep gaps within fragments to recharge the energy buffer. Passive RF communication is given as an example: fragments consist of transmitting a single bit. Another example is image sensing, where sleeps can be inserted between pixels and even within the different stages of sensing a single pixel.

The first question was about time scales. Pengyu answered that one charge-discharge cycle takes about 100µs, and that one image can be sensed in a few minutes. Somebody then asked how much energy could be saved by this technique. Pengyu replied that QuarkOS does not reduce energy consumption but extends the operating limits of sensors so that they can still execute tasks, albeit slowly, when limited energy is available. Mike Freedman (Princeton) asked at what scale QuarkOS can be applied. Is there a niche between battery-powered devices running conventional sensor OSes and micro-motes running without OS? Pengyu answered that their experiments used the Intel WISP architecture, which fits into this category. These devices have the advantage of being much easier to use than really small motes, where functionality needs to be embedded in hardware. John Ousterhout (Stanford) inquired about the limits of the power buffer. Pengyu explained that larger buffers are possible but disadvantageous: they require over-proportionally longer charge times, need more energy to reach the operating voltage, and cause more heat to be emitted during the discharge.

Open Mike

The open mike session started with Jonas Wagner (EPFL) asking whether partial information from low-rate video processing or low-bandwidth wide-area analytics is really more beneficial than the traditional case where users see full information or none at all. Ariel Rabkin replied that partial information is less scary than it sounds, and definitely useful.

The discussion continued around onloading vs offloading tasks to phones. There are many forms of offloading, some of which are well received. For example, Web sites can be fetched and rendered in the cloud, and be streamed to the phone at the right resolution.

Another topic that was raised was whether hardware could help with fragmenting tasks into even smaller units than what is possible with QuarkOS.

Be More Tolerant, but Not Too Tolerant

Summarized by William Jannen (wjannen@cs.stonybrook.edu)

Failure Recovery: When the Cure Is Worse Than the Disease

Zhenyu Guo, Sean McDirmid, Mao Yang, and Li Zhuang, Microsoft Research Asia; Pu Zhang, Microsoft Research Asia and Peking University; Yingwei Luo, Peking University; Tom Bergan, Microsoft Research and University of Washington; Madan Musuvathi, Zheng Zhang, and Lidong Zhou, Microsoft Research Asia

Zhenyu Guo began with an explanation of Microsoft Azure's leap day bug as an example of how efforts to recover from faults can actually do more harm than help. He analyzed service failures at major companies, and described three of several categories of common misbehaviors: resource contention, "recovered" software bugs, and service dependencies. Zhenyu argued that any failure recovery effort should be engineered to do no harm, because many of the bugs he described led to cascading failures that brought down many healthy system components when trying to recover from a small number of faults.

Zhenyu noted that one element commonly missing in failure recover design is systems thinking—the process of understanding how things interact with a system as a whole. Some decisions may seem correct locally, but are not necessarily globally correct. Systems thinking must be applied in all phases: design, testing, and deployment.

Petros Maniatis asked how easy it is to determine whether an action will do harm or not. Zhenyu explained that it is not easy, and that they have identified challenges in each step of the development cycle. There is no single solution that can solve all problems.

Someone posited the idea that systems thinking might result in a bunch of ground states that the system falls back into rather than cascading failures. In the context of the cloud, ground states might result in the cloud not processing jobs, and therefore not making money. A guiding principle might instead be

"don't lose money," rather than "do no harm." Risking cascading failures might be better than running the risk of not making money. Zhenyu agreed that this is a concern, but said that systems thinking is applicable in many situations.

John Ousterhout wondered whether the real problem was that error recovery code never gets debugged; it happens infrequently, but if developers knew it was there, they would fix it.

Toward Common Patterns for Distributed, Concurrent, Fault-Tolerant Code

Ryan Stutsman and John Ousterhout, Stanford University

Ryan Stutsman noted that many current applications scale to support billions of users, and developers write code that is distributed, concurrent, and fault tolerant. When managing thousands of logical threads of execution, the control flow must be adaptive and recover from failures easily, which impacts the way that programs are written. Developers have no control over when faults occur; traditional imperative code doesn't work, and execution history cannot be relied on. Ryan argues that it is only the state of the current system that really matters, and that programs should take steps based solely on state. While working on RAMCloud, they developed rules, tasks, and pools as a pattern for writing fault-tolerant code.

Ryan described rules, which are predicates based on actions. Actions fire in response to whatever conditions happen to be correct at the given moment. He explained that tasks group rules together with the state that they act on. Each task also has a goal, which is an invariant that the task is to achieve or maintain. Pools group tasks for a subsystem. In this pattern, execution order is determined by state instead of by some predefined ordering, and the execution order can adapt dynamically.

Mike Freedman noted that one way to think about this is that developers are designing systems that represent finite state machines. But it is more general than that, and you don't want to hard code a set of states; writing with this pattern should use actions and triggers. He wondered whether people using this model often write static state machines. Ryan responded that the patterns he's noticed have not had explicit state tags. The conditions apply implicitly. The model is not really about explicit states, but how to reason locally.

Peter Bailis wondered whether Ryan could compare their approach to rule-based languages like Bloom. Ryan was not familiar enough to speak about Bloom, but he thinks about the problem in a similar manner to how model checkers work: the programmer defines conditions and invariants.

John Wilkes observed that in practice, people actually write little state machines, and he thought that the idea of small-scale state machines applied lightly is a powerful idea. Ryan was concerned with the idea of explicit state machines for reasons of scalability. He would like to be able to reason about a system with just a local view of its state.

Escape Capsule: Explicit State Is Robust and Scalable

Shriram Rajagopalan, IBM T. J. Watson Research Center and University of British Columbia; Dan Williams and Hani Jamjoom, IBM T. J. Watson Research Center; Andrew Warfield, University of British Columbia

Shriram Rajagopalan noted that cloud infrastructure scales, and applications should be able to scale easily on that infrastructure as work increases. He proposed the capsule abstraction, a modification of applications and operating systems so that they support scaling at session granularity. The proposal would decouple sessions from applications; mobile sessions would allow balanced scale-out and scale-in, and replicated sessions would allow efficient and transparent fault tolerance.

Each layer must annotate the state that it wants to export, and each capsule must explicitly name its dependencies. A vertical chain of dependencies is called a "slice," which can represent the entire running state of a session. A centralized entity would be responsible for knowledge of capsules at each layer, and it would be able to unplug a slice, move it to another machine, and then plug the capsule back in at the destination. Shriram argued that elasticity and fault tolerance support should be provided at the system level, which the capsule abstraction provides.

Steve Muir commented that capsules were conceptually similar to Google's app engine, and he inquired about the tradeoffs of being intrusive. He noted that for many Web applications, the failure model is simply to drop the connection and restart. Shriram replied that if a single app engine is overloaded, there is no way to shed load dynamically and wait for the request to terminate. App engine scaling occurs at request boundaries.

Erez Zadok inquired as to which entity is responsible for detecting and setting dependencies. Shriram replied that the developer of every layer is responsible for setting dependencies and for registering the capsule. Erez followed up by asking about a case where there are many dependencies and inter-dependencies, to the point that it is cheaper to migrate the whole VM. Shriram noted that most session-based applications do not have dependencies that are so widespread.

Peter Druschel (MPI-SWS) noted that capsules were cheaper than process migration, but more intrusive. Historically, process migration has lost out in favor of VM migration, and Shriram was asked what made him think this trend would reverse. Shriram contended that there is a tradeoff; the coarser the granularity of migration, the less benefit in terms of fault tolerance and elasticity.

Timothy Roscoe asked which sessions would work well in the model. Some sessions might be hard to slice, and for sessions that are short-lived, there would be no point to migrating. Shriram said that for servers with millions of requests per second, this would not make sense, but that normal Web commerce applications have sessions that are not short-lived. A few minutes is more than enough time to overload a machine, and it is a

large enough window that a machine can fail, causing a loss of all session state.

Open Mike

The session began with a discussion of Ryan Stutsman's work. Petros Maniatis wondered about the case where two rules created an infinite loop, where each triggered the other. Ryan responded that there is no way to prevent programmers from writing infinite loops, but goal states help. If reaching a goal state takes too long, log messages are generated to help identify the problem. How one could ensure that atomic session code could be kept error free, specifically in the case of memory allocation failure, was also asked. Ryan responded that due to the expense of malloc, they mostly use preallocated buffers. He said that large external failures cannot be ignored, but local error handling can be done. Ariel Rabkin noted that a consequence of state machines being implicit is that it becomes difficult to ensure that progress is being made. Ryan commented that timers help, just as they help to detect infinite loops.

Erez Zadok shifted the discussion back to cascading failures. He noted that many of the examples from Zhenyu's talk suggested that a global view would allow better job handling and recovery. He noted that it might be difficult for a centralized controller to manage large systems, and wondered if a distributed version was considered. Erez likened the situation to current discussions in the world of electrical grid systems, where buildings or city blocks could disconnect themselves from the grid in the case of failure. Zhenyu replied that restricting failures to containers would help. He also noted that reusing existing failure detection mechanisms is useful.

The session concluded with further discussion of escape capsules and the difficulties that arise when retrofitting capsules to software stacks that were not designed with capsules in mind. Shriram noted that developers may not identify all state that needs to go into a session, and that plugging and unplugging capsules is not an easy job, especially in the presence of unpredictable processes like garbage collection.

Biiiig

Summarized by Seungyeop Han (syhan@cs.washington.edu)

Large-Scale Computation Not at the Cost of Expressiveness

Sangjin Han and Sylvia Ratnasamy, University of California, Berkeley

Sangjin Han presented Celias, a new programming model for large-scale computation. He started by reviewing the MapReduce family (including Dryad and Spark). Although those frameworks support bulk transformation of immutable data, they are not well suited to fine-grained updates on the data set. In their experiments with an iterative MapReduce job for k-hop reachability, they found that overhead takes more than 95% of the whole computation. Further, MapReduce cannot handle dynamic dataflows evolving at runtime. Sangjin proposed a new solution to fix those problems while preserving scalability and the fault tolerance properties of MapReduce.

Their programming model, Celias, is based on the classic programming model, Linda. Whereas Linda uses the process model and does not have any automatic scaling or fault tolerance features, Celias introduces microtasks as the computation model and uses tuplespace as data model. Microtasks are written as signature and code, and are triggered by the availability of tuples that match with the signature. The used input tuple is then automatically replaced by the output tuple. This programming model allows automatic scaling and fault tolerance without the intervention of programmers. Additionally, Sangjin noted that Celias is at least as expressive as MapReduce.

Matt Welsh (Google) commented that sometimes the immutable property is important, especially for rerunning as a batch, and it is important to find killer apps. Michael Freedman (Princeton) said that small tasks would kill performance with frequent I/O. John Ousterhout (Stanford) asked about the consistency issue. Sangjin answered that Celias is relying on atomic operations to ensure that updates are consistent. Petros Maniatis (Intel Labs) asked whether Optimus over Dryad would not solve the problem. Sangjin explained the approach is more like SQL and SQL query optimization and does not give the expressiveness that Celias provides.

When Cycles Are Cheap, Some Tables Can Be Huge

Bin Fan, Dong Zhou, and Hyeontaek Lim, Carnegie Mellon University; Michael Kaminsky, Intel Labs; David G. Andersen, Carnegie Mellon University

Bin Fan presented a new hash table that can serve a very large number of entries entirely from memory. Their target is when keys could be large whereas each value costs a few bits. He showed an example of the hash table storing UserID → online/offline. In the traditional hash tables storing those entries, some rows are not utilized. Additionally, storing keys to avoid collision takes another large space. Overall, it requires O(k+v) bits/entry.

By contrast, Bin's team suggested a new data structure to save memory. The core idea is to throw away the keys and to do brute force to avoid collisions. To do so, their algorithm, SetSeparation, enumerates hash functions in a hash function family to find the hash function that maps all keys in a group to correct values. Then, it records the parameter to get the hash function. Dividing the entire input into small groups, their scheme can handle a large number of keys/values. By the algorithm, their data structure uses only 0.5 + 1.5v bits/entry. Bin noted that the algorithm has a caveat that it cannot handle a membership function because it does not maintain keys by itself. In evaluation, SetSeparation uses only 3.88 MB for 16 million entries, whereas the STL (Standard Template Library) map uses 869.46 MB and the lookup speed is faster.

Jonas Wagner (EPFL) asked how Set Separation handles updates. Bin answered that it needs to keep track of which keys

are in the group in external storage. Volodymyr Kuznetsov (EPFL) commented that STL map is not a hash table and asked whether lookup and update cost would depend on key-size. Bin noted they were using a hash map and lookup is still constant although a little bit tricky. Michael Freedman (Princeton) asked whether figuring out which group the query key is in is not key-dependent. Bin replied that determining it is again based on hashing. Roxana Geambasu (Columbia) asked about concrete applications, noting that it has restrictions. Bin mentioned software routers as one example. Dan Williams (IBM Research) commented that it would be expensive for the cases with longer values. Bin said that it needs to be done per-bit for a multi-bit case, and the benefit decreases for longer values.

Wanted: Systems Abstractions for SDN

Sapan Bhatia, Andy Bavier, and Larry Peterson, Princeton University

Sapan Bhatia started by noting that iptables were functioning as a Swiss Army knife for many network configurations: while iptables is a powerful tool, it has the reputation for being tedious to use and error-prone. Additionally, changing configuration leads to resetting state, such as policies or routing entries. The research community has provided useful results, including new network architectures, domain-specific languages (such as Click), OS extensions, and finally SDN. In practice, however, nothing is changed and configuration still involves iptables.

Sapan explained that they have taken the best of academic ideas with standard tools. He presented NativeClick, which combines Click Modular Router's language to specify the graph and native runtime overlaid on the Linux networking stack. More specifically, elements and ports of Click are replaced with executable scripts and virtual links. Key mechanisms allowing this are from the network container to isolate route tables, policies, and virtual links. For connection to SDN, Sapan noted that expanding SDN to the end host is important. Also, he showed an SDN perspective consisting of vdev, controller, and processes in a middlebox.

Andrew Baumann (MSR) asked how to debug iptables since it requires understanding the Click abstraction. Sapan noted that it is an open problem in the generated codes, and current iptables itself is hard enough to debug. John Wilkes (Google) asked about evaluation. Sapan said that it is more community-driven, and users do not complain about it. Shriram Rajagopalan (UBC) commented that the SDN connection is a bit weak. Sapan noted it is about how you do middlebox functions and that the systems and the SDN approaches meet, since it achieves the end-result of SDN through OS functions.

Open Mike

The open mike session started with a question from Siddhartha Sen (Princeton) to Bin Fan about whether inserting lots of new keys could affect the performance. Bin answered that each group can handle a small number of keys, and thus more than 30 keys per group may require rehashing. Erez Zadok (Stony Brook University) continued with a comment that this is somewhat similar to Bloom filters and worth exploring the similarity. Bin replied that the difference is that their mechanism does not make any mistakes for the known keys, which a Bloom filter might do. One person from MSR wondered whether Bin's team used the same code for underlying hash functions in CHD (the Compress, Hash, Displace algorithm) when they evaluated. Bin answered they used the reference code from Google; a coauthor, Hyeontaek Lim (CMU), added that a number of entries would degrade CHD performance as well, and thus changing the underlying hash function would not change the trends.

There was a big discussion about applications for system research. Brian Noble (U Michigan) said that everyone should spend time finding someone doing computationally intensive projects. Timothy Roscoe (ETH) mentioned that computational finance and sociology will be interesting fields in terms of applications, and John Wilkes (Google) added biology and medicine. Petros Maniatis (Intel Labs) said that applications do not need to be solid ones, but it does make the work plausible. John Wilkes commented that for something big, we do not have an application yet, and we need to think not of applications, but problems and how we can solve them. Matt Welsh (Google) said that we have to get inspiration from problems out there and need to do generalization. Timothy Roscoe said that he had found someone with a big problem: he had teamed up with people who had fled the big banks and investment companies, as well as people still working at Credit Swiss, to do work on financial modeling. He has also worked with the Swiss Federal police in tracking counterfeited watches shipped around the world.

Someone commented that many people need help identifying their problems. Brad Karp (UCL) gave an example of block boundaries that are used for many other problems, although not for applications, but it is a fundamental problem of bigger systems.

Catching Up in the Clouds

Summarized by Deian Stefan (usenix@deian.net) and Edward Yang (ezyang@cs.stanford.edu)

The Case for Tiny Tasks in Compute Clusters

Kay Ousterhout, Aurojit Panda, Joshua Rosen, Shivaram Venkataraman, Reynold Xin, and Sylvia Ratnasamy, University of California, Berkeley; Scott Shenker, University of California, Berkeley, and International Computer Science Institute; Ion Stoica, University of California, Berkeley

In data-parallel computing, the straggler problem arises when a single task runs at a much slower rate (e.g., because it's running on a slow machine) than other tasks, slowing down the whole job. Yet, we typically schedule large batch tasks to ensure high cluster utilization. This not only amplifies the straggler problem, but also gives rise to another problem: cluster responsiveness. By running long batch tasks, short interactive jobs may need to wait on the order of seconds or minutes before being serviced, effectively rendering the cluster unresponsive.

To address these issues, Kay Ousterhout argued that all data-parallel jobs should be broken down into tiny tasks. This addresses the straggler problem by ensuring that workloads are evenly distributed across machines; fine-grained scheduling ensures that slow machines are assigned fewer tasks than fast machines. A simulation on Facebook workloads showed that using tiny tasks would improve the response time by roughly 5x. In a similar fashion, the tiny tasks paradigm bridges the gap between cluster utilization and responsiveness: long-running batch jobs are broken down into thousands of tiny tasks, allowing short interactive jobs to be interleaved as launched.

There are many challenges in implementing an architecture that employs the tiny task paradigm. To narrow the challenges, the authors focus on applying the model to data-parallel computations similar to MapReduce. In such a scenario, a task is typically I/O bound (reading input data stored on disk), and, to ensure high disk utilization, a tiny task must run for at least a few hundred milliseconds—a duration they argue that is acceptable even for Web applications. This is challenging as it requires changing the programming model to break a job into many tiny tasks, reducing the launch of a task to a few milliseconds, implementing a task scheduler that handles millions of decisions per second, and changing the underlying distributed file system to handle many small reads; however, using similar techniques to Spark and FDS, the authors believe they can address some of the concerns; developing a practical architecture, although promising, is part of their ongoing work.

Mike Schroeder (MSR) asked for a characterization of the jobs for which the straggler problem was not solved by their solution. Ousterhout noted that tiny tasks require a change in the programming model, but programmers can ignore this and, for example, can still write code that contains infinite loops—in such cases, tiny tasks won't do much to improve the situation. Jeff Mogul asked how long a job should be, as opposed to how long it can be (i.e., short enough to read 8 MB as to use the disk efficiently). Ousterhout noted that the few hundred milliseconds is consistent with the shortest duration of data-analytics jobs they've observed in practice. Hyeontaek Lim pointed out that dividing a 40,000-task job into 4 million won't necessarily be "better"; what size jobs should be sub-divided? Ousterhout explained that they had looked into the space to find characteristics of different jobs and found that jobs with a few tasks were the ones with long-running tasks; finding the precise point where diving into more tasks becomes inefficient is part of future investigation.

Using Dark Fiber to Displace Diesel Generators

Aman Kansal, Microsoft Research; Bhuvan Urgaonkar, Pennsylvania State University; Sriram Govindan, Microsoft

High availability is a lot of work. A server may be protected against power failure by a UPS; but this is no good if your network gateway goes down: datacenters must also install diesel generators to protect against utility failure; but this, too, fails in the event of physical disaster, so your data must be georeplicated. Highly available services are deployed with multiple layers of redundancy, and this redundancy is expensive. Because high availability services must always be georeplicated, Aman Kansal suggested relying solely on georeplication for availability, reducing the availability needs for any given datacenter.The authors argue that "Geo-distributed Bunches of Datacenters" (or GBoDs) could be practical, but there are a number of questions to answer. For one, how much can one reduce DC availability before global availability is affected? Assuming independent failure, one can calculate this out: for n=10, one can do with 0.1% failure probability rather than 0.001%. A bigger question is how applications need to adapt to this new scheme. Some methods of georeplication, such as sharding distributed state, no longer work as everything must be replicated everywhere—addressing this is an open research problem. Bandwidth, however, is not a problem: the authors propose that the dark fiber connecting these datacenters be used to carry out the large amounts of data transfer necessary to perform full replication.

Timothy Roscoe pointed out that building a new datacenter takes a really long time: on the order of seven months, which is quite different from spinning up a new server. Jeff Mogul noted that as the reliability of single datacenters decreases, the error bars on your availability calculation increase. One might do OK if there is an error margin built into your availability figures; but that margin costs money, exactly what GBoDs are trying to save. Edouard Bugnion asked which workloads could be distributed this way, and Aman answered that without software redesign, read-only software is the only thing that can be done; applications with real-time data writes are considerably more difficult.

Towards Elastic Operating Systems

Amit Gupta, Ehab Ababneh, Richard Han, and Eric Keller, University of Colorado, Boulder

Amit Gupta said that one of the main benefits of cloud-based systems is the ability to elastically change the amount of resources allocated to an application according to demand; however, we presently place the burden of elasticity on apps: an app has to, a priori, be designed to operate in a cloud environment. The developer must design the app such that it can distribute the workload, on demand, among different instances; handle data consistency issues (e.g., sharing across instances); and monitor load as to decide when to expand or contract the number of nodes.

Rather than continue building apps with elasticity in mind, Gupta argued for making elasticity an OS primitive. ElasticOS would allow applications to be built without any notion of elasticity, while transparently expanding and contracting to accommodate different workloads. To this end, they propose using elastic page tables, i.e., page tables that map virtual addresses to machine/physical addresses, as a way to allow an application to expand when memory on other nodes becomes available and is

in demand. Different from previous distributed shared memory (DSM) systems, they, however, do not replicate data pages across machines. Instead, paging-in remote tables results in them being moved from the remote machine. This avoids the need for coherency protocols that have plagued DSM systems; however, to take advantage of locality, they propose migrating the process/thread execution context once the number of pages that are being pulled in reaches a certain threshold. Unlike data pages, this can be quite efficient because caching multiple copies of code pages does not require DSM-like protocols. Gupta concluded the talk with the remark that although various issues (e.g., fault tolerance and elastic network I/O) need to be addressed, their preliminary Linux implementation has shown promising measurements.

Jay Lorch (MSR) was skeptical about the approach, as it wound up leading researchers on the same path as DSM. In response, Gupta noted that their work differs from the DSM efforts in two important ways: DSM heavily relied on replication and kept execution context fixed (except for process migration); in their work, they keep a unique copy of data and move execution contexts when appropriate. Andrew Warfield noted that moving contexts around is expensive (because it requires transferring roughly a page of context information) and asked why moving the context to the data is a good idea (because this happens often when stretching to a large number of nodes). Gupta noted that they adopted a hybrid approach: they pull data until they notice that they can exploit locality, and at that point they jump. He further noted that for certain workloads this approach may not work, but this requires further investigation. Timothy Roscoe brought up the issue of memory efficiency: if code pages are replicated to allow fast context transfers, at what point does this approach become inefficient? Gupta noted that in data-intensive applications, such as MySQL, the number of code pages is much lower than the corresponding number of data pages, so they do not anticipate a large overhead if code is carefully replicated across a (part of the) datacenter. The last questioner asked whether there is any reason to believe that cluster-wide parallelization is going to be better than multicore. In response, Gupta noted that a process on a single node is inherently bound by memory and they intend to break that barrier.

Open Mike

Rik Farrow provided the quote of the session: "I think you live in an alternate reality called Google."

Everyone seemed to agree about tiny tasks for cluster computing (except one guy from Berkeley), so the conversation turned to a discussion about GBoDs and elastic computing.

The subject of datacenters was close to the heart of many of the industrial members of the audience. Two interesting topics came up during the ensuing discussion. The first was political reasons why applications may not be georeplicated; for example, a country may have strict data privacy laws that prevent data from being replicated across its borders. Jeff Mogul mentioned that this was exactly the case, and that they had implemented selective georeplication. John Wilkes (Google) brought up the cost calculation that companies are constantly doing when considering datacenter administration. Some infrastructure has 11 datacenters deployed to serve 10 datacenters' worth of load, with the last datacenter running compute jobs on the extra capacity. As opposed to infrastructure such as Google AppEngine, which has excessive redundancy, GBoDs may not be a win in such situations. Additionally, when a datacenter goes down, there is the cost of all the hardware that is not being utilized in that datacenter; one participant noted that making sure that this hardware is not wasted is worth at least some money.

The response to the elastic computing talk had been considerably more prickly, and so Jeff launched a new discussion by pointing out that ElasticOS was targeted at being fully backwards-compatible, whereas tiny tasks and datacenters asked programmers to change their programming model. "Aren't we underestimating the value of not changing applications?" Matt Welsh responded that at Google, "We are constantly changing our applications to adopt new programming models." This led to Rik Farrow's response: "I think you live in an alternate reality called Google." There was some debate whether or not MapReduce was an example of a new programming model that had been rapidly taken up by non-Google programmers. Lim countered by stating that Hive/Pig were used by people who looked at MapReduce and said, "We want SQL." Depending on who you ask, the majority of MapReduce jobs are written in these languages.

Others were confused about whether or not ElasticOS bought anything in an era where machines with 1 TB memories could be purchased. Moving around all this data, especially in a failure tolerant way, would be difficult. "At some point," one participant commented, "won't brute force just win out?" The authors acknowledged this, and argued that you'd have to make locality assumptions about the usage of 1 TB of memory.

Correct, Secure, and Verifiable

Summarized by William Jannen (wjannen@cs.stonybrook.edu)

Toward Principled Browser Security

Edward Yang, Deian Stefan, John Mitchell, and David Mazières, Stanford University; Petr Marchenko and Brad Karp, University College London

Deian Stefan noted that the Web has evolved into an application platform. And although traditional operating systems provide applications with page protection and file system permissions, the browser must rely on the same origin policy (SOP) to protect data. There are exceptions to strict isolation in the SOP; on the one hand, these exceptions allow developers to build complex, information-sharing apps; on the other hand, exceptions can lead to leaks of sensitive data.

Deian listed several remedies for SOP shortcomings, such as the content security policy (CSP) and cross-origin resource sharing

(CORS), but noted that such measures are coarse-grained, static, and inflexible. He proposed a more principled approach—to use information flow control (IFC) as a browser security primitive. Browser-based IFC would do more than just emulate the SOP; it would allow execution of untrusted code on sensitive data. A strict base policy could enforce origin non-interference, but the framework would allow flexibility and fault isolation.

Matt Welsh asked about the proposal's implications on both browser and Web API designs, and whether it would require a change to all browsers and all API code. Deian noted that the proposal would require browser modifications, but it would not require a modification of JavaScript; it would be just another API that developers could use. Deian was then asked about memory and performance overheads, and the potential implications that overheads would have in the browser performance war. He replied that although he did not have numbers on hand, there would be no impact on the performance of existing code. The proposal is effectively an opt-in and coarse-grained approach. Don Porter requested some implementation insights. Deian responded that it is implemented as a whole new API. They leverage Gecko's compartment model, with all implementation done at the language level.

Deian was asked to discuss the differences between their proposal and FlowFox from CCS. He explained that the FlowFox mechanism was for JavaScript only, was not opt-in, and could break existing Web sites; also, it does not support declassification. Ashvin Goel (U Toronto) asked how to ensure that attackers could not simply bypass checks, especially in the presence of browser bugs. Deian noted that avoiding bugs is difficult, but that they leverage Gecko's compartment model to isolate memory spaces.

Volodymyr Kuznetsov (EPFL) asked about side channels. Deian commented that this is an extension of their previous work that does address some side channels, but with respect to external timing channels there is not much they can do. Peter Bailis asked whether an opt-in policy would allow adversaries to hide in legacy content. Deian clarified that the proposal would not impose on existing Web sites, but a Web site that uses the API would be protected.

-OVERIFY: Optimizing Programs for Fast Verification

Jonas Wagner, Volodymyr Kuznetsov, and George Candea, École Polytechnique Fédérale de Lausanne (EPFL)

Jonas Wagner noted that there are many tools that prove the safety and correctness of software, but that these tools are rarely used in practice because often they are slow or hard to use. One reason that existing tools are slow is because they receive the wrong kind of input—a performance-optimized binary; the time it takes to verify a program can be made significantly faster by compiling specifically for verification instead of for execution on a CPU. As an example, branches are costly for verification, and equivalent branch-free code often can be verified more easily.

Jonas proposed a compiler switch to enable verification optimizations, much in the way -g is used for debugging, and -O3 for performance. The -OVERIFY flag would signal the compiler to preserve high-level information, favor optimizations that ease verification, annotate the program, and generate runtime checks so that verification tools can easily detect bugs. They actually have an implementation that they have tested.

Ariel Rabkin asked whether performing these optimizations inside the compiler or inside the verification tool itself makes more sense. He also wondered whether verification time was really a limiting factor. Jonas noted that the time it takes to verify is important; a drastic cost reduction would not only save developer time, it could change the ways that verification tools were used, to the extent that they potentially could be used at every commit. And one of the principal advantages of using a compiler flag is that it does not require any changes to existing verification tools.

Ariel then asked if the same tweaks are valuable for all verification tools. Jonas explained that there are different types of tools; their prototype, -OSYMBEX, generates code optimized for symbolic execution tools. Martín Abadi then posed an idea: what if a compiler could generate several different versions of the binary, each optimized for verifying a particular property? Jonas noted that this would work particularly well for finding concurrency bugs.

When Andrew Birrell (MSR) asked about high-level information that can't be transferred down to assembly, Jonas remarked that a binary with debugging information has complete source code, but that not all information is necessary. High-level types, and information about which variables are local, global, or thread local would be helpful.

Global Authentication in an Untrustworthy World

Martín Abadi, Andrew Birrell, Ilya Mironov, Ted Wobber, and Yinglian Xie, Microsoft Research

Andrew Birrell gave a quick recap of authentication with X.509 certificates, noting many positive features: authentication is completely decentralized, non-hierarchical, and worldwide. Additionally, X.509 is pervasive and quite secure; however, Andrew pointed out that being quite secure is almost as bad as not being secure at all. He used a few high-profile examples of failures to prove this point. The underlying problem is the large scale of trust—the relying party trusts every CA in the delegation chain, not just the root or the leaf. Intermediate CAs are all uniformly powerful and can write a certificate for any name. Andrew argued that although non-hierarchic authentication is essential, uniform trust of worldwide CAs does not work. Local policies are a better approach.

Andrew then discussed the details of their data set. A 2010 EFF data set was parsed and then supplemented with additional data collected in 2012. In total, 7.8 million certificates were acquired from 22.7 million TLS handshakes, and the details were

organized in an SQL database. Although the database enables ad hoc queries, the data is too large for ad hoc analysis; they instead performed cluster analysis, choosing a set of 18 features that were thought to be interesting, including key length, country, trusted root, etc. The result was a set of 28 tight clusters with few outliers.

Andrew presented uses for the data set, such as a user-controlled policy engine. The database could be queried to make trust decisions. Policies could be designed by experts and selected by the end user.

Mike Freedman wondered why SPKI never took off, given that it allows chained delegation. Andrew responded that SPKI allowed Web-of-trust-like things, but clearly there was not enough demand. People seem quite happy with the current situation using X.509, except that it breaks two times per year. Deian Stefan asked about data access. Andrew hoped that Microsoft would allow the data set to be made public, but he noted that the 2010 EFF data set is available.

Petros Maniatis asked about the implementation of any policies that might have made sense for Microsoft, and whether Andrew had evaluated how many Web sites had such policies "turned off." Andrew joked that had they done this evaluation, it would have been an SOSP paper, but they are currently working on it.

Automated Debugging for Arbitrarily Long Executions

Cristian Zamfir, Baris Kasikci, Johannes Kinder, Edouard Bugnion, and George Candea, École Polytechnique Fédérale de Lausanne (EPFL)

Cristian Zamfir explained that the debugging process, identifying and fixing the root cause of a program failure, differs during development and production. During development, the gdb record option can be used to reverse step from the point of failure, but in the production world, a core dump from a segmentation fault cannot be reverse-stepped. Although production level record support is possible, overheads may be prohibitive. The question, then, is what can be done with limited information in production systems?

Cristian proposed reverse execution synthesis (RES), which takes as input a program and its core dump, and outputs an execution suffix that would lead to that core dump. He noted a key insight is that there exists a large class of programs for which the root cause is close to the actual point of failure, making the search space manageable; however, the challenge is inferring the paths. This can be done by recording constraints through branches and checking against the core dump state. By applying this process recursively, the system can build an incrementally larger execution suffix. As long as the start of the path contains feasible values, the execution suffix is guaranteed to reach the error state. RES can debug arbitrarily long programs with no runtime overhead.

Steve Hand wondered how many distinct paths were often observed. Cristian replied that RES works well for small concurrent programs, and that they are able to synthesize unique suffixes in about a minute. But, in general, a program that overwrites much of its state would result in many execution suffixes.

John Wilkes asked whether logs could be leveraged. Cristian said that logs could provide path information, which is important. They would not provide full paths, but they would provide specific points, which could disambiguate state.

Petros Maniatis asked about the tradeoffs of checkpointing at runtime, and then combining forward and backward search. Cristian replied that fast checkpointing might be something worth using and could potentially be used to validate the feasibility of states. But his position is to do as much as possible without recording; checkpointing is a form of recording.

Jeff Mogul asked whether the compiler could be leveraged, like -OVERIFY, to generate log entries at specific points where reverse stepping would be difficult. Cristian said that the compiler could try to use less overwriting, and that they are trying to use copy-on-write when possible.

When John Wilkes asked for project insights, Cristian replied that the project is still in its beginnings. Execution suffixes are currently on the order of hundreds of instructions, but it depends on the specific program and how much rewriting it does. He noted that without debugging symbols, a control flow graph is necessary in order to determine possible paths.

Open Mike

George Candea wanted to know how comfortable people were with putting specialized code in programs solely for post-mortem analysis. He was curious about the range of measures with which people were comfortable. Matt Welsh wanted clarification as to whether George was asking about developers, libraries, or runtimes. George responded that that was the point of his question. He thought some people might be uncomfortable with a 5% overhead, but Matt thought that 5% was absolutely fine because the information gained was invaluable. John Wilkes noted that monitoring systems generate several percent overhead, so overheads under one percent are well within the acceptable threshold. Jeff Mogul said that what is unacceptable is logging information that causes privacy concerns.

Matt Welsh asserted that reviewers should make sure to avoid punishing papers when the overheads are over these thresholds. Also pointed out is that just because a technique is not acceptable for production, it is still worth reading. Mike Freedman commented that it is also important for authors to be careful about how they calculate overheads. Erez Zadok reiterated that acceptable costs are dependent on the application. NASA's Jet Propulsion Labs might be willing to accept overheads of 20–30% for a Mars rover, so the community shouldn't set simple thresholds. John Wilkes added that thinking about the cost to fix bugs is also important. There should be more flexibility than

just one magic number. What we would like is a range of things and different choices. Overheads accumulate, so thinking about priorities and making sure that important features are the ones that are ultimately incorporated is important; what might be acceptable on a server might not be acceptable on a phone.

Petros Maniatis asked about the role of hardware. He noted that Intel provides branch information such as last branch records (LBR), but that in terms of performance, these things are not free. Intel must prioritize things, too, so if the software community would come to a consensus, then hardware designers could make these decisions.

A general comment was that the session's debugging papers assumed a C-code environment, but there also is interest in managed language runtimes. A lot of production code is written in languages such as Java and C#, and this might be an easy place to add diagnostics. An open question was how general can these tools be made.

Something Old, Something New, Something Hot

Summarized by Cristian Zamfir (cristian.zamfir@epfl.ch)

Operating System Support for Augmented Reality Applications

Loris D'Antoni, University of Pennsylvania; Alan Dunn and Suman Jana, University of Texas at Austin; Tadayoshi Kohno, University of Washington; Benjamin Livshits, David Molnar, Alexander Moshchuk, and Eyal Ofek, Microsoft Research; Franziska Roesner, University of Washington; Scott Saponas, Margus Veanes, and Helen J. Wang, Microsoft Research

David Molnar explained that augmented reality (AR) applications impose new challenges on operating systems for several reasons. First, AR applications must deal with potentially sensitive data that gets mixed with user input, which calls for a more fine-grained permission system. David showed how the raw video input stream may contain user faces and private information, yet any application can access this information, so this will not work with AR applications that multiplex access to the same video stream. Second, the window system will have to be updated in order to handle 3D objects from multiple applications, as opposed to the square windows we have today. Third, AR systems have to deal with continuous inputs (e.g., gestures) that are also inherently noisy (e.g., an object may be confused with an arm).

David pointed out that given the emergence of such systems, these challenges (especially the privacy-related ones) will have to be solved before the legislation is updated in probably 2–3 years. Otherwise, without some privacy guarantees, AR systems may even be officially banned from certain contexts.

Michael Freedman (Princeton) asked what lessons from Web mash-ups can be applied in this area. David mentioned that the work on clickjacking defense can be used. Another issue is the Same Origin Policy, which does not yet exist in AR systems, but there is room to innovate in this area.

Steve Muir (VMware) asked if the OS should manage the access to private data. David argued positively, and briefly described his upcoming paper in USENIX Security on how to provide visual explanations to users of what the requested permissions allow applications to access. Stefan Bucur (EPFL) asked whether information flow control could help. David agreed that is a good direction for exploration. Peter Druschel (MPI) asked whether there will be a "one size fits all" set of abstractions for the AR applications. David said that the answer is likely yes, since this model will be easier to use by developers.

Solving the Straggler Problem with Bounded Staleness

James Cipar, Qirong Ho, Jin Kyu Kim, Seunghak Lee, Gregory R. Ganger, and Garth Gibson, Carnegie Mellon University; Kimberly Keeton, HP Labs; Eric Xing, Carnegie Mellon University

James Cipar introduced Stale Synchronous Parallelism, a model that maps to scientific applications and can tolerate stragglers. The key idea is that this model allows applications to tolerate significant delays in some threads. Preliminary results with an early prototype show that increased staleness can mask the effects of occasional delays. The model also detects when data becomes too unsynchronized, and synchronizes threads to avoid unbounded staleness. An important open question for ongoing work is how to automatically tune the requirements of the application regarding freshness.

Doug Terry (MSR) asked whether the staleness bound impacts convergence and James answered that, in their experience, it is important. Mike Schroeder (MSR) asked whether their method works with non-transient delays. James answered that their approach supports temporary delays, like a GC pause or some additional computation done by a specific thread, but it cannot do anything against non-transient delays. Roxana Geambasu (Columbia) asked what other kind of applications this model accommodates. James said they have experience with scientific computing applications, page rank, and machine-learning algorithms that resemble gradient descent. Jonas Wagner (EPFL) asked why performance improves when there are no delays. James answered the staleness model masks some delays. David Ackley (UNM) pointed the authors to related work that uses a similar technique to tolerate transient errors. This technique works for errors, but might apply also to delayed computation.

Lightweight Snapshots and System-Level Backtracking

Edouard Bugnion, Vitaly Chipounov, and George Candea, Ecole Polytechnique Fédérale de Lausanne (EPFL)

Edouard Bugnion introduced the concept of lightweight snapshots, a new state abstraction that provides immutable snapshots integrated into the virtual memory subsystem. Based on the lightweight snapshots abstraction, he proposed a design for an operating system that provides system-level backtracking for arbitrary applications. The design of the backtracking OS leverages modern x86 hardware-virtualization support to perform efficient backtracking and supports configurable scheduling policies.

Edouard gave several examples of applications that can benefit from the backtracking OS (e.g., S2E, a demanding application that implements full-system symbolic execution, and Z3, an SMT solver). He also exemplified the system-level backtracking API using the canonical n-queens example. Their early prototype can already provide backtracking capabilities to complex applications such as Z3, with minimal changes to the application.

David Molnar (MSR) asked whether developers can pass the scheduling heuristic to the OS. Edouard answered this is indeed possible. Andrew Bauman (MSR) asked whether it would be better to move the scheduler outside the OS. Edouard answered that the scheduling policy and the scheduler are decoupled: the scheduler can be in the OS, and the scheduling policy can be set by the application. Edward Yang (Stanford) asked whether the proposed abstraction can be thought of as a faster fork(). Edouard answered that it is more than that, since it is hard to just use fork() and combine it with various search heuristics. Brad Karp (UCL) asked whether privilege separation (as in Wedge, a system built at UCL) is another application of the proposed system. Privilege separation requires strong isolation, but can this be added? Edouard answered that Wedge was eventually built into Dune. The big takeaway is that one can now envision building domain-specific operating systems.

HAT, Not CAP: Towards Highly Available Transactions

Peter Bailis, University of California, Berkeley; Alan Fekete, University of Sydney; Ali Ghodsi, University of California, Berkeley and KTH/Royal Institute of Technology; Joseph M. Hellerstein and Ion Stoica, University of California, Berkeley

Peter Bailis proposed highly available transactions (HATs) that are available in the presence of network partitions. The CAP theorem shows that it is impossible to provide linearizability in the presence of arbitrary network partitions, and does not directly apply to database transactions. Peter pointed out that even single-node databases do not provide serializability by default, because it is expensive. Instead, they provide weaker consistency models, and many applications work well with these models and can tolerate the arising anomalies to gain performance. However, it is not clear which models can be achieved with high availability.

Their work is about exploring the class of high availability low-latency transactions that can be achieved in the presence of network partitions. Peter proposed techniques based on read or write buffering to provide some guarantees (read committed and repeatable read isolation) for a HAT system, and also described some additional guarantees that they proved are not achievable (e.g., regency bounds and some integrity guarantees).

Brad Karp (UCL) noted that previous papers about Spanner and Eiger mentioned similar social networking examples (e.g., the order of the posts). Brad asked what HAT can provide compared to this other work. Peter answered that there are many existing applications that work with the weak consistency offered by today's databases, so this is a useful programming model. Moreover, the anomalies that would appear under these models do not appear for some applications. For instance, TPCC isn't subject to anomalies from weak consistency, which is why Oracle is TPCC-compliant and offers a weak consistency model. Doug Terry (MSR) argued that one way to implement repeatable reads is to just not allow any transactions to commit when you have a partition. Peter said that with transactions you can have success and abort, so one can abort everything and obtain the liveness property. Their paper contains details on how they define transaction availability. Michael Freedman (Princeton) asked whether the write buffering technique is two-phase commit. Peter answered no and explained the differences.

Open Mike

Byung-Gon Chun (Microsoft) asked how the bounded staleness model compares to the asynchronous lazy synchronization model used in GraphLab. James answered that GraphLab makes assumptions about data locality and would also require modifications to their algorithms to accommodate staleness. Petros Maniatis (Intel) asked whether their work is about figuring out how much staleness can be supported by the applications. James answered that they established a profile of the applications that work, and identified several applications that fit the profile. Steve Hand (Cambridge) suggested that if one speculates, then one may also need to roll back, so they could use lightweight snapshots proposed in the talk by Edouard Bugnion.

Jacob Lorch (MSR) asked how to evaluate which of the consistency models discussed in the HAT not CAP talk is reasonable and can be understood by users. Peter Bailis (Berkeley) argued that it is still an open question what consistency models to run on and not violate the application's integrity constraints. Peter argued this is a great direction that should see more work and exemplified with work from Marc Shapiro at INRIA on conflict-free replicated data types. Siddhartha Sen (Princeton) proposed comparing the code that one would have to write to deal with weaker vs stronger consistency. Ali Ghodsi (Berkeley) commented that Doug Terry's session consistency model already prevents several anomalies that users see, so the big open question is what is the consistency model that is both efficient and prevents most of these anomalies.

Hardware to the Rescue

Summarized by Cristian Zamfir (cristian.zamfir@epfl.ch)

The von Neumann Architecture Is Due for Retirement

Aleksander Budzynowski and Gernot Heiser, NICTA and University of New South Wales

Gernot Heiser's talk was motivated by the plateau reached by CPU frequency and the multicore trend; he proposed a self-modifying data flow graph computation model to replace the von Neumann model. Their model essentially does away with global memory, thus aiming at making it possible to express and

implement general purpose parallel computations easier and more efficiently.

A typical data flow computing model is static, and there is no way to express dynamic algorithms and data structures. To address this challenge, they propose a data flow graph that can change itself, change references to other nodes in their immediate neighborhood, create new nodes, etc. They have a partial implementation that takes Haskell code as input and translates it into data-flow assembly.

Ariel Rabkin (Princeton) wondered how synchronization is implemented and asked to see how the proposed design works for something simple like matrix multiplication. Gernot answered that synchronization is entirely done by data flow. He also mentioned that the example he described in the talk is more complex than a matrix multiplication and would work for dynamic data structures. Mike Schroeder (MSR) asked about the next step; where do they plan to get the hardware to implement this? Gernot said they can try to simulate this architecture in software without the performance benefits. Moreover, their work is inspired by a startup that aims to build fully asynchronous hardware.

David Ackley (UNM) said that the answer to all the open questions raised by the talk is coming up with a spatial layout of the graph, which has to be embedded in the hardware, which has to be spatially extended, yet still be finite. Gernot answered that there is commonality between their hardware and the hardware proposed by David at the previous HotOS. They are trying to get away from the global address space yet retain as much of the CS abstractions as possible, thus making the model more easy to program than David's model.

Brad Karp (UCL) asked whether before proposing such a change at the hardware level, one does not have to refute the arguments made by people working on taking a sequential programming model and making it work for multicores. Gernot argued that everyone is trying to tweak the von Neumann model, but these approaches will run out of steam after some scale. He argued that his system has some nice properties that are worth exploring.

Arrakis: A Case for the End of the Empire

Simon Peter and Thomas Anderson, University of Washington

Simon Peter argued that recent hardware devices enable building kernels that allow applications to talk to hardware directly, without OS mediation; the kernel only provides control plane services (e.g., deals with resource reallocation), but applications use a library linked in their address space to talk to hardware directly. One enabler for this design is the fact that hardware is increasingly virtualized. Moreover, I/O devices become faster while CPUs are bottlenecked by frequency, so unmediated access to hardware devices is an important performance-related requirement.

One of Arrakis' several goals is to allow applications to customize OS functionality (e.g., provide protection domains using hardware protection). Moreover, Arrakis is designed to provide device driver safety, by running device driver replicas and ensuring that when one replica crashes, the system does not crash. One important challenge is dealing with the fact that hardware may not provide sufficient virtualization capabilities for meeting all the proposed design goals.

Jeff Mogul (Google) said Arrakis looks like it is partially reinventing the InfiniBand model (which has had this separation for a decade). Simon answered they are trying to generalize that model to other hardware. Steve Muir (VMware) argued that Arrakis needs to support migration and checkpointing to be useful for real-world use cases and Peter agreed. Edouard Bugnion (EPFL) asked what can be learned from the way people build the control/data plane separation in network hardware. Simon answered this was part of their inspiration and that they are already looking at that literature.

Rethinking Network Stack Design with Memory Snapshots

Michael Chan, Heiner Litz, and David R. Cheriton, Stanford University

Michael Chan proposed a redesign of the network stack, which leverages HICAMP (ASPLOS '12), a hardware memory system that supports snapshot isolation. The system allows zero-copy, reduces memory allocations, and works with the existing socket API. The main motivation for this work is that the networking stack uses many memory allocations and accesses, while network I/O speeds are going up. Unlike existing approaches, users do not have to use specific data structures to do zero-copy; instead they can use the application data. Compatibility with the POSIX API is done by simply passing another flag to the malloc() call to use HICAMP memory.

Michael showed how to do zero-copy I/O and how to simplify the DMA process and the NIC design. He also discussed the space and time tradeoff of the design. He ended the talk by arguing that software-hardware co-design can improve OS architecture and solicited ideas for applications to other areas of system design.

Siddhartha Sen (Princeton) pointed out that persistent data structures (some developed by Targent) can be used to efficiently keep multiple copies of a data structure and be able to update it partially. Jacob Lorch (MSR) asked when the hardware will be available. Michael mentioned they have a simulator and plan to make it available to others soon.

Rik Farrow (USENIX) mentioned that their system ends up doing pointer chasing, which imposes some overhead. Michael said there are two additional reads/write when writing duplicate data. Michael mentioned some back-of-the-envelope calculations for network I/O that seem very optimistic (several hundred Gbps), so even achieving 50% of that would be impressive.

Edouard Bugnion asked about the downside when integrating with the cache hierarchy. Michael answered that L3 will take care of most of the caching for their data structures, but in L1 and L2 would only contain immutable data, so there is no need to maintain cache coherency. He envisioned a selector that can be configured to tell the CPU whether the range needs to be handled by the HICAMP controller or the CPU.

Open Mike

Steve Muir (VMware) asked if the approaches discussed can be partially implemented (e.g., implement memory snapshots for just for a part of the memory). Gernot Heiser argued against sacrificing the purity of the model, otherwise the model will never take off. Michael argued that you can use HICAMP as an accelerator, not a replacement for paged virtual memory, so they advocate a hybrid model. Simon Peter argued that for Arrakis they do not advocate a hybrid model, but one could retrofit Arrakis onto KVM, for instance.

Jonas Wagner (EPFL) commented that the discussed hardware models seem to map very well for some workloads, but not for all, and asked whether there are systems with little workload diversity for which these systems would work well. Several attendees gave examples of systems that run dedicated workloads (e.g., OLTP) that could benefit from the proposed hardware changes (e.g., snapshots). Jonas also gave an example for functional languages that could implement reference counting more efficiently in hardware. Gernot agreed that functional languages map very well to a data flow model. Simon also argued that garbage collection also maps very well. Jacob Lorch and Eduard Bugnion suggested that hardware-software co-design is a fascinating area for innovation, but we should not rely only on hardware people to design hardware, otherwise the hardware is hard to exploit. Some examples are hardware that can help do efficient garbage collection and hardware that can efficiently demultiplex. Simon said an open question is what happens if the hardware is not flexible enough at demultiplexing: can a software solution be found?

David Molnar (MSR) pointed out a new piece of hardware that looks interesting: tritium batteries that do not require charging. An open question is how to re-architect the OS assuming such new hardware.

Unconference Results

Summarized by Rik Farrow (rik@usenix.org)

Hardware's Role in System Design

Michael Chan presented the summary of what I thought of as Petro Maniatis' session about the future of CPU and system designs. He pointed out that Intel is swayed by what it expects its biggest customers will want in the future, and what systems researchers want. Software writers want better performance, but also better views of the internal metrics collected by processors. Power consumption is one of Intel's biggest focuses right now, but there are also issues of hardware and software mismatch. For example, Barrelfish relies on cache coherency for inter-core communication, but this works poorly for data structures (or anything larger than six cache lines). Finally, software folks struggle to imagine what will come out of the Intel CPU pipeline five years down the road, the current timeframe for integrating changes in CPUs, and secret by design.

Networking CPU Cores

Jeff Mogul presented a summary of John Ousterhout's unconference session, which was focused on John's desire for a high-speed network that would connect CPU cores and their level 1 caches together with very low latency. The conclusion was that switch designers have already worked on a very similar issue, exchanging packets of data across a switch fabric with very low latency, and that John should talk with the people familiar with these designs. Jeff pointed out that John doesn't want queues, but Jeff said that there must be queues.

Augmented Reality and Mobile Sensors

David Molnar (Microsoft) first thanked Franzi Roesner (U Washington) for helping lead this session. Then he explained what is different in new settings, such as Google Glass and more immersive augmented reality (AR) displays: the input and the output. The input is noisy, sounds and video, and much of it should be private. The output must be controlled, so that malicious apps don't overlay reality with their own version—for example, rewriting a sign. The OS must create a permissions experience and abstractions to control what applications can access which data. We no longer have 2D windows, but 3D volumes. AR makes several existing problems much worse.

There are issues of privacy as well, such as bystander privacy, or places that want a complete ban on video recording, like a gym or a bar in Seattle. There are also man-in-the-middle concerns, such as a government that seeks to collect data on its citizens. David suggested having primacy of the physical space—for example, allowing the owner of a space to zap a camera using an infrared laser. He concluded by saying that there is about a two-year window to deal with this before legislatures start mangling these issues.

Programming Language Approaches to Systems

Edward Yang (Stanford) began by pointing out that programming language and software can be codesigned, and you can even build a language just for yourself. They discussed composition and modularity, the ability to have many languages that can work together. They want incrementalism, which means backward compatibility and no flag days, but also the ability to exclude what doesn't work well. Ed mentioned the difficulty in measuring programmer productivity, and concluded by saying that program languages people should be hired, as they often bring useful insights into projects.

Security

The security unconference group was one of the largest, but the ground covered seemed all-too familiar to me. Deian Stefan (Stanford) presented the summary. The group began by considering a trust model for code integrity, then pondered allowing untrusted code to modify or copy data. They posited that they know how to isolate untrusted code, and that the interesting question is how to share data between sandboxes. They next considered machine learning for security, and whether authentication (actually authorization) should be considered on a scale.

They also considered the role of firewalls in security today, concluding that firewalls provide insufficient protection and that getting them to provide better protection would require a huge amount of user interaction. Plus, firewalls do not protect against internal attackers. They ignored the issue that the attacker who has established a beachhead through the typical spearphishing attack is essentially an insider. This negates having a firewall in almost all of the attacks on organizations seen today.

They finished their session by discussing the role of the user in making security decisions, asking whether they can educate non-power users about security. Restructuring designs that avoid requiring the user to make any security decisions was the final point (and a very good one). My apologies for the editorial comments, and while I only witnessed the end of the discussion, I found myself disturbed by hearing old ground covered while summarizing the notes for the entire session.

Big OLTP: Oxymoron or Impending Crises

Using a graphical reference to Oracle, Peter Bailis (UC Berkeley) began the summary for this session with a question: when will the current tech we use break? Peter said that OLTP follows two common patterns: low mutation rate with many queries, or lots of mutation but few queries. And with devices like Google Glass, there will be both high mutation and lots of queries. Closed-world assumptions about databases will no longer hold, with the source of truth being external to the stream processor. They expect to see OLTP combined with OLAP (analytics), and the challenge will remain providing isolation between queries (ACID).

Big Data Analytics

Byung-Gon Chun presented 13 slides, the most thorough and the longest summary. He began with six slides where the group attempted to define big data, and presented a nice sound bite: the three Vs of Volume, Velocity, and Variety. While volume is clear enough when speaking of big data, and velocity obviously refers to the ability to process that data swiftly, variety means that data may be unstructured.

The group came up with eight areas of interest. The first was low latency, i.e., the ability to work interactively, to recognize significant events in data, and to remain efficient as the volume of data grows. Second was data management, which refers to the issues of data labeling, data format (e.g., HDF5), standardization, provenance, and new data structures. Unified execution is a simple concept: being able to process data on a single box or a scaled-up cluster using the same program. The fourth issue, related to unified execution, is unified programming. Spark and Hive were presented as examples. Workflow management was the fifth issue, the ability to schedule and coordinate a set of related jobs, along with tools for doing this.

Their sixth issue was resource management, which implies at least prioritization or constraints that control how many resources a job can use. While an economic approach was suggested, it was also pointed out that Cosmos, a chargeback scheme, is not working. The seventh issue was accuracy, in the sense that sometimes approximate answers, requiring less processing, are acceptable, and there needs to be the ability to adjust the desired accuracy. The final point was configuration complexity, with Hadoop being used as a bad example, having tens of configuration parameters. What is needed is auto-tuning knobs, where the knobs set desired goals instead of tweaking specific parameters.

Elastic OS

Amit Gupta, who presented a paper about elasticity in operating systems, convened this unconference session to further explore the issue. The participants wondered whether an ElasticOS for generic processes is too broad a goal, but perhaps certain applications, or even threads, would be suitable for elasticizing. Elasticizing may occur for different reasons, even shrinking a process when resource costs go up and expanding when costs go down, and the process could use more resources. In the end, the group concluded that they still need to be convinced.

Verification

Ariel Rabkin (Princeton) organized this session, wrote a summary, but left before he could present it. On his slides, he had written that they now believe that increasingly large artifacts can be verified if the artifact was designed with verifications in mind. Formalization of code design is possible, probably usable, but is only cost-effective for safety-critical code, and not usable yet for Web companies.

HotPar '13: 5th USENIX Workshop on Hot Topics in Parallelism

San Jose, CA
June 24-25, 2013

Panel

Tools in the Real World

Summarized by Rik Farrow (rik@usenix.org)

Panelists: Niall Dalton, Calxeda; Brandon Lucia, University of Washington and Microsoft Research; Tipp Moseley, Google; Paul Peterson, Intel Corporation

Brandon Lucia has just gotten his Ph.D. from the University of Washington and is going next to MSR. Brandon started talking about software development tool research. Development tools eat data, such as programming traces and source code. Next, we need to abstract the data (for example, convert program traces

to event traces). Abstractions helps us facilitate analysis, the final step, for example, in suggesting a solution for a problem in performance.

Brandon provided a concrete example from his own work, a project called Recon (recon.cs.washington.edu), for concurrency debugging. Recon uses CPU hardware to monitor shared data accesses, uses this to build context-aware graphs, and analyzes these graphs to reconstruct the root cause of a failure.

Brandon ended by covering some trends. Statistical modeling and analysis allows you to take big piles of data and make sense out of them, distilling the data into a model. The next trend is the collection of data in real time, such as instrumenting all of Google's servers to capture rare events in situ. Third, tools can also be used for automation, not just for analysis but also for fixing problems. The last trend Brandon talked about was closing the gap between hardware architecture and software tool designers. Hardware support allows you to collect data that you wouldn't otherwise be able to collect.

Tipp Moseley began by saying that tools solve problems. Google collects hundreds of thousands of profiles every day, including hardware counters (instructions per second, branch misprediction, cache misses) and software profiles (heap size, growth, lock contention, disk fragmentation). They process this data to produce reports on potentially anomalous results for applications, libraries, and even functions. Because Google owns the entire stack, every time you submit a change, your change includes tests so that the change can be analyzed. Tipp said that their tools work well for uncovering race cases, while some other tests, like load tests, are difficult to test. Google does profile applications in production, but scale is a huge problem. A one-in-a-million race condition will happen all the time at Google's scale. Static analysis works poorly at this scale, because the systems are so large with many interacting programs on distributed systems.

Tipp said that the really hard problems cross boundaries. For example, each Web request comes in through load balancers, to frontends, to backends, then to storage. It becomes very difficult to figure out where a problem occurs in this chain, discovering what causes long tail latency, for example, in performance.

Tipp wants tools that have low overhead, such as sampling that takes less than 3%, as well as more hardware counters.

Niall Dalton said the most important tool is coffee. Niall displayed a chart on which there is a latency spike every 500 ms after an OS upgrade, and asked how we would solve this. In another example, a new version of a system comes in, and again there are latency spikes that show up routinely, but software tools fail to discover what's causing the problem. Niall explained that the problem lay in the BIOS, and that he had to hack the BIOS to fix the problem. Both examples were single applications on dedicated systems. Niall next described having two applications on the same box, both stressing RAM access, but tools that trace applications wouldn't see that. Niall said changes to disk seek patterns, network incast, and the effects of big data applications that are not on the system under observation but affect its performance are like a "whale swimming by." So you have your own problems, plus your neighbors'.

A lot of us have built ad hoc tools over the years, but the hardest problem is to discover where, deep in the system, something is going on. Just think of dueling schedulers. And it might take 2,000 hours before a kernel crash occurs.

Paul Peterson said that when you are in the software tools business, your software will work better on your hardware than on other hardware. Paul added that he was speaking for himself, not Intel. Although people are most familiar with Intel as a hardware company, Intel has also been a software company that has been working in the world of parallelism since multicore CPUs became common. Intel works with BIOS, device drivers, operating systems for optimizing performance, and with 14,000 engineers worldwide.

Paul works on the Parallel Studio suite of products, focusing on the node level, but also on the cluster level, with tracing and analysis tools at each level. For example, Advisor XE helps people design and build parallel programs. They also have Composer, Intel MPI, VTune amplifier, and Inspector, which looks for memory leaks. Intel produces enabling software that helps developers.

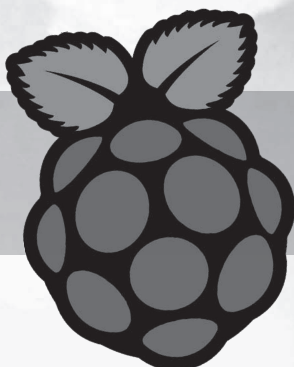
The chair started off by saying that coffee is his favorite tool, too. He then asked Tipp whether some of the concerns he has are Google-only problems, that is, only large data companies have these problems. Tipp said that in 10 years, everyone is going to have to deal with them, even on smartphones. Another panelist said that cloud computing is already producing environments that look like a lot of problems within Google. Paul ranked his top three list of customer complaints: tools shouldn't break (especially debugging tools—broken debuggers really piss people off); speed matters (for anything other than hardware tools) and overhead should be less than 10%; and finally, the tool doesn't produce enough data. People want tools to be faster and richer. Niall said that these problems already exist, say, if they want traces on a group of systems instead of one.

Next, the chair asked Tipp what they did to solve race detection. Tipp said he didn't solve this himself, but that much smarter people built tools built on Valgrind that just seem to work. A lot of the work is based on fine-tuning edge cases. Google has good test coverage, but doesn't have good tools for doing race detection on code working at production scale. Brandon said that he didn't think that race detection problems are solved, that the overhead is too high (10x). Tipp said that he wanted that side, the production side, solved as well.

Note: The compete reports from HotPar '13 are available online at www.usenix.org/publications/login

Coming Soon!

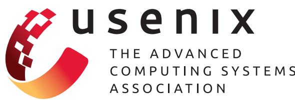
A brand new magazine for the Raspberry Pi Community



Look for us at your local newsstand

UK/EU Sept. 21 US/Can Oct. 18 Australia Nov. 18

Or find us online at www.raspberry-pi-geek.com



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

Lucky LISA '13

27th Large Installation System Administration Conference

NOVEMBER 3–8, 2013 • WASHINGTON, D.C.

Keynote Address: "Modern Infrastructure: The Convergence of Network, Compute, and Data"
by Jason Hoffman, *Founder, Joyent*

Join us for **6 days of practical training** on topics including:

- **SRE Classroom: Non-Abstract Large System Design for Sysadmins** by John Looney, *Google*
- **Root Cause Analysis** by Stuart Kendrick, *Fred Hutchinson Cancer Research Center*
- **PowerShell Fundamentals** by Steven Murawski, *Stack Exchange*
- **Introduction to Chef** by Nathen Harvey, *Opscode*

The **3-day Technical Program** includes:

- Plenaries by Hilary Mason, *bitly,* and Todd Underwood, *Google*
- Invited Talks by industry leaders such as Ariel Tseitlin, *Netflix;* Jeff Darcy, *Red Hat;* Theo Schlossnagle, *Circonus;* Matt Provost, *Weta Digital;* and Jennifer Davis, *Yahoo!*
- Paper presentations, workshops, vendor exhibition, posters, Guru Is In sessions, BoFs, and more!

New for 2013: The LISA Lab Hack Space!

www.usenix.org/lisa2013