;login:
APRIL 2015 VOL. 40, NO. 2

usenix
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

Sysadmin and Distributed Computing

usenix 40TH ANNIVERSARY

usenix
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

UPCOMING EVENTS

NSDI '15: 12th USENIX Symposium on Networked Systems Design and Implementation

May 4–6, 2015, Oakland, CA, USA
www.usenix.org/nsdi15

SREcon15 Europe

May 14–15, 2015, Dublin, Ireland
www.usenix.org/srecon15europe

HotOS XV: 15th Workshop on Hot Topics in Operating Systems

May 18–20, 2015, Kartause Ittingen, Switzerland
www.usenix.org/hotos15

USENIX ATC '15: 2015 USENIX Annual Technical Conference

July 8–10, 2015, Santa Clara, CA, USA
www.usenix.org/atc15

Co-located with USENIX ATC '15 and taking place July 6–7, 2015:

HotCloud '15: 7th USENIX Workshop on Hot Topics in Cloud Computing
www.usenix.org/hotcloud15

HotStorage '15: 7th USENIX Workshop on Hot Topics in Storage and File Systems
www.usenix.org/hotstorage15

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

USENIX Security '15: 24th USENIX Security Symposium

August 12–14, 2015, Washington, D.C., USA
www.usenix.org/usenixsecurity15

Co-located with USENIX Security '15:

WOOT '15: 9th USENIX Workshop on Offensive Technologies
August 10–11, 2015
www.usenix.org/woot15

CSET '15: 8th Workshop on Cyber Security Experimentation and Test
August 10, 2015
Submissions due April 23, 2015
www.usenix.org/cset15

FOCI '15: 5th USENIX Workshop on Free and Open Communications on the Internet
August 10, 2015
Submissions due May 12, 2015
www.usenix.org/foci15

HealthTech '15: 2015 USENIX Summit on Health Information Technologies
Safety, Security, Privacy, and Interoperability of Health Information Technologies
August 10, 2015
www.usenix.org/healthtech15

JETS '15: 2015 USENIX Journal of Election Technology and Systems Workshop
(Formerly EVT/WOTE)
August 11, 2015
www.jets-journal.org

HotSec '15: 2015 USENIX Summit on Hot Topics in Security
August 11, 2015
www.usenix.org/hotsec15

3GSE '15: 2015 USENIX Summit on Gaming, Games, and Gamification in Security Education
August 11, 2015
Submissions due May 5, 2015
www.usenix.org/3gse15

LISA15

November 8–13, 2015, Washington, D.C., USA
Submissions due April 17, 2015
www.usenix.org/lisa15

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:
APRIL 2015 VOL. 40, NO. 2

usenix
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

EDITOR
Rik Farrow
rik@usenix.org

MANAGING EDITOR
Michele Nelson
michele@usenix.org

COPY EDITORS
Steve Gilmartin
Amber Ankerholz

PRODUCTION
Arnold Gatilao
Jasmine Murcia

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.

©2015 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

A frontal attack on idleness. That's how I see distributed systems: a way to keep systems as busy as possible. Decades ago, computers were much too expensive to be allowed to sit idle. Today, computers are immensely more powerful, less expensive, and often set up in enormous clusters, but we still need to keep them busy.

Letting a modern system sit idle wastes resources, even if CPUs have gotten more efficient at doing nothing. Hard disks will continue to spin, DRAM will still require dynamic refreshing, and the power supplies and other hardware will continue to turn electricity into heat, whether a system is busy or not. Certainly a busy system will use more energy, but not a lot more.

And all the while a system is sitting there idle, it's becoming obsolete. No wonder there has been such a focus on distributed systems.

History

While researching for this issue, I took a look at the table of contents for Andy Tanenbaum's 1994 edition of *Distributed Systems*. I was quickly reminded that the distributed systems that come to mind today have their roots in the past, going back as far as the diode and tube-based DYSEAC in 1954. The DYSEAC not was actually distributed, but the potential was there and was discussed. The DYSEAC was perhaps the first mobile computer, housed inside a truck.

We need to skip ahead many years before we begin to see functioning distributed systems. While there were parallel systems built earlier, the earliest commercial distributed system was the Apollo AEGIS operating system in 1982, later Apollo Domain. Apollo used Ethernet to share storage and provide security, processing, and even signed software updates. Sun's "the network is the computer" occurred because Sun was attempting to build an Apollo-like system on top of BSD UNIX.

By the end of the 1980s, two well-known research distributed systems had been implemented: Amoeba and Sprite. Both presented single system images, in that a person sitting at any terminal could be using resources on any of the connected systems transparently. Sprite could even migrate processes from one system to another, while both systems used network file systems.

In the early 1990s, largely as a result of the UNIX wars [1] that began when Sun and AT&T decided to unite SunOS (BSD 4.2 to 4.3) and System V UNIX, the Distributed Computing Environment (DCE [2]) was born. The bastard offspring of Apollo Domain (then owned by HP) and the Andrew File System, DCE was designed to provide many of the features of Apollo Domain, but be able to do so across a network consisting of systems from different vendors. Although the Wikipedia page mentions just three major components, getting DCE to run required running six different services, starting them in the correct order, and was not easy to do. And there was no single reference implementation, so with each vendor writing its own version, interoperability was only experimentally achieved.

The Common Object Request Broker Architecture (CORBA [3]) actually has been more successful than DCE. CORBA supports data sharing and remote processing using an interface definition language (IDL) to create abstractions of objects or data structure in

many programming languages. The use of an IDL goes back to work Sun did on remote procedure calls (and likely earlier) as a mechanism for sharing data portably. And each participating host needs to run just one service, the request broker, making it simpler to operate than DCE.

Today, we have truly massive distributed systems. These systems use Java programs as their middleware, something that really puzzled me when I first heard of this. Why Java, I thought, when earlier distributed systems, like Amoeba and Sprite, were microkernel-based for performance and efficiency?

MapReduce focuses on the processing of bulk, line-oriented data, where the latency of disk I/O is the big limiting factor, not the processor. The ease of programming in Java greatly outweighed any performance issues through the use of Java. Systems like Cassandra are very successful, and stable, partially because of their reliance on Java.

Idleness

And what does any of this have to do with idleness? In the cases of Amoeba and Sprite, a workstation user could enter a command that would be executed on any available system. Amoeba included a parallel Make (Amake) for compiling in parallel, the better to use resources.

For datacenters, the cost of power and cooling are second only to the costs of servers [4], so keeping those servers busy maximizes efficiency. As servers will only last three to five years before being replaced, letting them sit idle is another form of waste.

There are other ways of using idle servers, the most popular being clouds running VMs. In some ways these too are distributed systems, as they are relying on distributed storage, authorization, and control systems. But cloud systems are not considered distributed systems.

The Lineup

We begin this issue with two articles related to distributed computing. Matthew Grosvenor, Malte Schwarzkopf, Ionel Gog, and Andrew Moore preview their NSDI paper [5] about a technique and software they've developed to reduce network latency for latency-sensitive applications. They first explain the problems caused by queueing, examine other solutions, explain how their simple solution works, then provide some data to back this up. I think their software may be a hit in datacenters.

Ben Pfaff and a large group of people from VMware write about some software that is already a hit. Open vSwitch [6] is the most widely used virtual switch in cloud environments, and runs on all major hypervisor platforms. The authors describe the architecture of vSwitch and techniques that have been used to both improve performance and increase efficiency of this open source software.

I decided to interview Andy Tanenbaum for this issue. Andy has published two editions of his very successful book about distributed systems, but what I was initially interested in was Amoeba, a distributed system designed by Andy and his students in the late '80s. I also wanted to take advantage of the interview to ask some questions about some of his other projects, and to get his opinion on the acceptance of microkernels.

We also have four articles focused on aspects of system administration. Tim Bradshaw leads off with an article about the living dead. These zombies are old systems that you cannot stop supporting even when the official support for the platform has long been deceased. Tim contrasts nimble and fast-growing organizations with ones burdened with decaying body parts, and what this means for system administrators and security.

I convinced Adam Moskowitz to write about his experience creating a testing framework for shell scripts. Adam does a great job making clear why you don't just want, but need, testing for all your scripts. Adam also shares the framework he built for automating script testing as much as possible. If you find yourself disturbed at the thought of having to add testing to your scripts, I suggest you read Adam's article. Perhaps you will see the light, as well as appreciate the toolset Adam recommends.

Andrew Stribblehill and Kavita Guliani take on the topic of managing incidents. Quite simply, you get to manage incidents, instead of managing crises, when you have previously prepared and practiced how to do so. Stribblehill and Guliani contrast unmanaged incidents (crises management) with managed incidents, and provide the elements of a managed incident process.

Andy Seely shares his own generational theory of sysadmins. I fit into the category that his father does (the first generation of explorers), but I do see his point. Like any system of categorization, Andy's system will not be a perfect fit for anyone, but I found his system to be a useful way of understanding the different types of sysadmins you will encounter.

David Blank-Edelman makes browsers dance in his column. David describes how to use the Selenium framework to remotely control a browser from Perl, allowing you to create repeatable tests for software that will be interacting with Web browsers.

Dave Beazley takes issue with how too many Python programmers handle exceptions. Dave criticizes much of the extreme coding he sees, from over-catching exceptions to ignoring them, while providing really practical advice.

Dave Josephsen expands his discussion of Graphios, a tool that he briefly mentioned in his February 2015 column. Dave has been working with the creator of Graphios (Shawn Sterling) to replace Graphios' Graphite-specific backend with a modular framework, so Graphios can extract data from Nagios and share it with other tools.

Dan Geer borrows from biological survey techniques to suggest analysis techniques that can help us decide whether patching is really improving software over time. While most of us wonder how patching cannot be helping (by removing exploitable code), Dan's focus is on whether the patching is making any difference to the overall security of a specimen of software.

Robert Ferrell takes a humorous look at both system administration and distributed systems. While you might imagine these would be difficult topics for humor, Robert manages to do a good job of it.

USENIX is celebrating its fortieth year, and we are adding some new features to *;login:*. Peter Salus will be writing for *;login:* again, reviving his history column with the story of the UNIX wars. We also have the very first issue of *UNIX News,* the predecessor of *;login:*. Finally, we have included another article from our archives. David Brumley, now an associate professor at CMU, wrote about the rootkits he encountered while working at Stanford University.

We have just two book reviews this month, both by Mark Lamourine. We also have a handful of reports from LISA14.

Conference Reports

Speaking of reports, we, the USENIX Board and staff, have decided that we are going to handle reports differently in the future. In the past, we have worked hard in an attempt to cover every session of every conference, symposium, or workshop that USENIX runs, and have done a fairly good job. While we occasionally managed to cover all sessions—for example, at a well-attended, single-track conference like OSDI—conferences like LISA, with its five tracks, have always been difficult to cover.

We also found ourselves competing with the sound and video recordings made of the sessions. USENIX has been recording sessions for many years now, and providing those recordings to anyone interested as part of our commitment to open access to research. Actual recordings of a presentation are much more accurate than reports, as they reproduce what actually happened instead of the summarizer's version of the presentation. While I personally attended many sessions, I lacked the ability to be in many places at once, no matter how many tracks were happening concurrently. I would use my notes to improve some of the reports I received, but will confess that by the end of two (or three or more) days of note taking, my notes were getting a bit sketchy.

In the future, we will continue to solicit and publish reports that we receive when they meet our standards. We will not go to the lengths we have in the past to round up summarizers in an attempt to cover every session, including those (the majority) that are being recorded. As a result, future issues of *;login:* may be a bit thinner. But USENIX will continue to provide audio and video recordings of as many sessions as possible.

The attack on idleness continues to this day, with even smartphones getting into the act. Most any Web program expects to execute code (JavaScript) with the browser, as well as code on the server. Bitcoin mining malware authors have been using distributed systems since 2011, and in 2014, five apps were removed from Google Play because they were attempting to use the comparatively puny processing power of smartphones to mine Bitcoins [7]. Malware that targets desktops with power to spare has been modestly more successful as conscripted miners.

The Internet of Things will continue the movement toward distributed systems, with each Thing doing more than simply collecting and forwarding data, like a FitBit or a Nike shoe. To help with this process, and to leverage the large number of programmers at home in Windows environments, Microsoft has announced that they will be releasing a version of Windows 10 for the Pi 2 [8]. While I welcome the news, I do feel a bit of trepidation, and hope that this is a securely stripped-down version of Windows. In the future, even Things will be kept busy.

References

[1] UNIX wars: http://en.wikipedia.org/wiki/Unix_wars.

[2] DCE, pretty decent reference: http://en.wikipedia.org/wiki/Distributed_Computing_Environment.

[3] CORBA: http://en.wikipedia.org/wiki/Common_Object_Request_Broker_Architecture.

[4] James Hamilton's blog, Perspectives, September 18, 2010: http://perspectives.mvdirona.com/2010/09/18/OverallDataCenterCosts.aspx.

[5] Matthew P. Grosvenor, Malte Schwarzkopf, Ionel Gog, Robert N. M. Watson, Andrew W. Moore, Steven Hand, and Jon Crowcroft, University of Cambridge, "Queues Don't Matter When You Can JUMP Them!"; NSDI '15: https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/grosvenor.

[6] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Joe Stringer, Pravin Shelar, Keith Amidon, Martin Casado, VMware, Inc., "The Design and Implementation of Open vSwitch"; NSDI '15: https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/pfaff.

[7] http://www.theregister.co.uk/2014/04/25/yes_there_is_now_bitcoinmining_malware_for_android/.

[8] Windows 10 for Raspberry Pi 2: https://dev.windows.com/en-us/featured/raspberrypi2support.

Letter to the Editor

I read with great interest your interview with Dan Farmer. A strong character, who loves to create things, but wants no part of "productizing"! His tale about Symantec and outsourcing security products made me wonder how that company assesses the honesty of its wares and brought to mind the genre of spam that offers to help rid one's machine of malware.

The piece also brought to mind some bits of ancient history:

COPS definitely wasn't the earliest vulnerability scanner, though it may have been the earliest to be publicly distributed. In fact the README file that comes with COPS says that sysadmins had been in the habit of rolling their own bits and pieces of it before. For one, Fred Grampp—who together with Robert Morris senior wrote a classic paper on UNIX security—had distributed a quite comprehensive security-sniffing suite within Bell Labs; unfortunately, I can't remember its name. The company's first computer-security task force, which I chaired in 1982, was able to back up its warnings with real data, thanks much to Fred.

I well remember the Morris worm. A bunch of people gathered in the research UNIX lab, watching it beat on the gate as we followed its progress by phone contact with other sites across the country. Peter Weinberger, in particular, spent much of the time on the phone with CMU's Software Engineering Institute, whose CERT division today proudly says, "We were there for the first Internet security incident and we're still here 25 years later." That incident led to the formalization of CERT's national role, which in turn provided Dan's first job.

The worm didn't get into Bell Labs, because Dave Presotto recoiled from installing the inscrutable Sendmail system and decided to roll his own. Then, in a reverse twist, he named it after a poison tree. If Dave put into Upas anything like Sendmail's trojan horse (a back door intended for diagnosing troubles reported by confused sysadmins), nobody has found it yet.

Doug McIlroy
doug@cs.dartmouth.edu

Jump the Queue to Lower Latency

MATTHEW P. GROSVENOR, MALTE SCHWARZKOPF, IONEL GOG, AND ANDREW MOORE

Matthew P. Grosvenor is a PhD student at the University of Cambridge Computer Laboratory. His interests lie in cross-layer optimizations of networks, with a particular focus on network latency. He has completed research internships at NICTA (Sydney), Microsoft Research Silicon Valley, and Microsoft Research Cambridge, and he maintains strong ties to the high-speed networking vendor Exablaze.
matthew.grosvenor@cl.cam.ac.uk

Malte Schwarzkopf is currently finishing his PhD at the University of Cambridge Computer Laboratory. His research is primarily on operating systems and scheduling for datacenters, but he dallies in many a trade. He completed a research internship in Google's cluster management group and will join the PDOS group at MIT after graduating.
malte.schwarzkopf@cl.cam.ac.uk

Ionel Gog is a PhD student in the University of Cambridge Computer Laboratory. His research interests include distributed systems, data processing systems, and scheduling. He received his MEng in computing from Imperial College London and has done internships at Google, Facebook, and Microsoft Research.
ionel.gog@cl.cam.ac.uk

In this article, we show that it is possible and practical to achieve bounded latency in datacenter networks using QJump, an open-source tool that we've been building at the University of Cambridge. Furthermore, we show how QJump can concurrently support a range of network service levels, from strictly bounded latency through to line-rate throughput using the prioritization features found in any datacenter switch.

Bringing Back Determinism

In a statistically multiplexed network, packets share network resources in a first come, first served manner. A packet arriving at a statistically multiplexed ("stat-mux") switch (or router) is either forwarded immediately or forced to wait until the link is free. This makes it hard to determine how long the packet will take to cross the network. In other words, stat-mux networks do not provide *latency determinism*.

The desire to retrofit latency determinism onto Internet Protocol (IP) stat-mux networks sparked a glut of research in the mid-90s on "Quality of Service" (QoS) schemes. QoS technologies like DiffServ demonstrated that coarse-grained *classification* and *rate-limiting* could be used to control Internet network latencies. However, these schemes were complex to deploy and often required cooperation between multiple competing entities. For these reasons (and many others) Internet QoS struggled for widespread deployment, and hence provided limited benefits [1].

Today, the muscle behind the Internet is found in datacenters, with tens of thousands of networked compute nodes in each. Datacenter networks are constructed using the same fundamental building blocks as the Internet. Like the Internet, they use statistical multiplexing and Internet Protocol (IP) communication. Also like the Internet, datacenter networks suffer from lack of latency determinism, or "tail latency" problems. Worse still, the close coupling of applications in datacenters magnifies tail-latency effects. Barroso and Dean showed that, if as few as one machine in 10,000 is a straggler, up to 18% of user requests can experience long tail latencies [2].

Unsurprisingly, the culprit for these tail latencies is once again statistical multiplexing. More precisely, congestion from some applications causes queueing that delays traffic from other applications. We call the ability of networked applications to affect each others' latencies *network interference*. For example, Hadoop MapReduce can cause queueing that interferes with memcached request latencies, causing latency increases of up to 85x.

The good news is that datacenters are also unlike the Internet. They have well-known network structures, and the bulk of the network is under the control of a single authority. The differences between datacenters and the Internet allow us to apply QoS schemes in new ways, different and simpler than the Internet does. In datacenters, we can enforce a system-wide policy, and, using known host counts and link rates, we can calculate specific rate limits that allow us to provide a guaranteed bound on network latency.

We have implemented these ideas in QJump. QJump is a simple and immediately deployable approach to controlling network interference in datacenter networks. QJump is open source

Andrew W. Moore is a Senior Lecturer at the University of Cambridge Computer Laboratory in England, where he is part of the Systems Research Group working on issues of network and systems architecture. His research interests include enabling open-network research and education using the NetFPGA platform. Other research pursuits include low-power energy-aware networking and novel network and systems datacenter architectures. andrew.moore@cl.cam.ac.uk

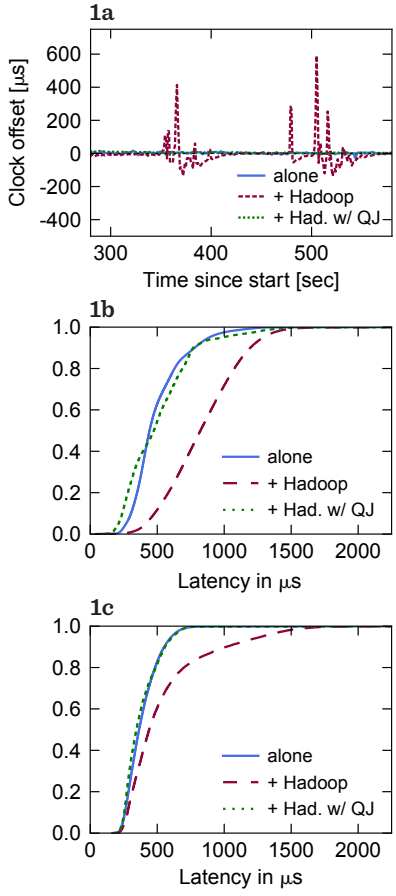


Figure 1a shows a timeline of PTPd synchronization offset. **Figure 1b** has a CDF of memcached request latency, and **Figure 1c** has a CDF of Naiad synchronization time.

and runs on unmodified hardware and software. A full paper describing QJump will appear in the 12th USENIX Symposium on Networked System Design and Implementation (NSDI '15) [3]. Additional information including source code and data is available from our accompanying Web site: http://www.cl.cam.ac.uk/research/srg/netos/qjump.

QJump in Action

To illustrate how bad network interference can get and how well QJump fixes it, we show the results from a collection of experiments with latency-sensitive datacenter applications (see Figure 1). In each experiment, the application: (1) runs alone on the network, (2) shares the network with Hadoop MapReduce, and (3) shares the network with Hadoop, but has QJump enabled. A complete evaluation of QJump, including full details of these experiments (and many others), is available in the full paper.

1. Clock Synchronization. Precise clock synchronization is important to distributed systems such as Google's Spanner. PTPd offers microsecond-granularity time synchronization from a time server to machines on a local network. However, it assumes roughly constant network delay. In Figure 1a, we show a timeline of PTPd synchronizing a host clock on both an idle network and when sharing the network with Hadoop. In the shared case, Hadoop causes queueing which delays PTPd's synchronization packets. This causes PTPd to temporarily fall 200–500 µs out of synchronization, 50x worse than on an idle network. With QJump enabled, the PTPd synchronization remains unaffected by Hadoop's traffic.

2. Key-Value Stores. Memcached is a popular in-memory key-value store used by Facebook and others to store small objects for quick retrieval. We benchmark memcached using the memaslap load generator and measure the request latency. Figure 1b shows the distribution of request latencies on an idle network and a network shared with Hadoop. With Hadoop running, the 99th percentile request latency degrades by 1.5x from 779 µs to 1196 µs. Furthermore, around 1 in 6,000 requests takes over 200 ms to complete, over 85x worse than the maximum latency on an idle network. With QJump enabled, these effects are mitigated.

3. Big Data Computation. Naiad [4] is a framework for big data computation. In some computations, Naiad's performance depends on low-latency synchronization between worker nodes. To test Naiad's sensitivity to network interference, we execute a synchronization benchmark (provided by the Naiad authors) with and without Hadoop running. Figure 1c shows the distribution of Naiad synchronization latencies in both situations. On an idle network, Naiad takes around 500 µs at the 99th percentile to perform a four-way synchronization. With interference, this grows to 1.1–1.5 ms, a 2–3x performance degradation. With QJump running, the performance nearly exactly conforms to the interference-free situation.

These experiments cover just a small set of applications, but there are many others that can also benefit from using QJump. Examples include coordination traffic for Software Defined Networking (SDN), distributed locking/consensus services, and fast failure detectors.

Scheduling and Queueing Latency

To understand how QJump works, we first need to understand the two main sources of latency nondeterminism in statistically multiplexed (stat-mux) networks: scheduling latency and queueing latency. In Figure 2a, a collection of packets (P) arrive at an idle switch S0. At

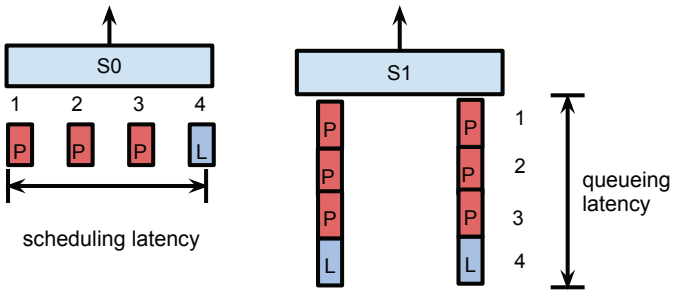


Figure 2: Latency causes (a) fan-in, packets waiting to be serviced by the switch scheduler, or (b) queueing, packets waiting behind many other packets.

the same time, a *latency sensitive* packet (L) also arrives. The L packet experiences *scheduling latency* as it waits for other P packets to be serviced by the switch scheduler. Scheduling latency is a consequence of *fan-in,* which happens when multiple packets contend for the same output port on the switch. If the switch takes too long to output packets, then new packets can queue behind existing ones. Figure 2b shows two latency sensitive packets (L) queued behind many other waiting packets (P). This is a kind of head-of-line blocking that we call *queueing latency*. Queuing latency is caused by excessive scheduling latency. We cannot eliminate scheduling latency in a stat-mux network. However, using some simple math, we can put a bound on it. By doing so, we can ensure that packets are issued into the network at a rate that prevents them from queueing up behind each other, thus also control queueing latency.

Bounded Queues—Bounded Latency

Considering Figure 2a, in the worst case the L packet will need to wait for the switch scheduler to service all preceding P packets before it is serviced. For a switch with n ports, the worst-case waiting time is $n - 1$ (approximately n) packets. As the number of ports on the switch grows, the worst-case latency grows with it.

We can easily expand this understanding to cover multi-hop networks by treating the whole network as a single "big switch" (this is an application of the "hose-constraint" [4] model). Hence we can apply the same calculation as above. Knowing that a packet of size P will take P/R seconds to transmit at link-rate R, we can therefore bound the maximum interference delay at:

$$worst\ case\ end\text{-}to\text{-}end\ delay \leq n \times \frac{P}{R} \qquad (1)$$

where n is the number of hosts, P is the maximum packet size (in bits), and R is the rate of the slowest link in bits per second. Equation 1 assumes that hosts have only one (active) link to the network and that the speed at the core of the network is never slower than the speed at the edge. We think that these are both safe assumptions for any reasonable datacenter network.

We refer to the worst-case delay as a *network epoch*. A network epoch is the maximum time that an initially idle network will take to service one packet from every sending host, regardless of the source, destination, or timing of those packets. Intuitively, if we imagine the network as a funnel, the network epoch represents the time that the funnel will take to drain when it is filled to the top. If all hosts are rate-limited so that they cannot issue more than one packet per epoch, no permanent queues can build up, and the end-to-end network delay bound will be maintained forever. That is, we rate-limit hosts so that the funnel will never overflow.

The problem with a network epoch is that it is a global concept. To maintain it, all hosts need to agree on when an epoch begins and when it ends. It would seem that this requires all hosts in the network to have tightly synchronized clocks. In fact, network epochs can work even without clock synchronization. If we assume that network epochs occur at the same frequency, but not necessarily in the same phase, the network becomes *mesochronous*. This requires us to double the latency bound, but all other properties hold (see [3] for further details). The network epoch thus becomes:

$$network\ epoch = 2n \times \frac{P}{R} \qquad (2)$$

Equation 2 is the basis for QJump. QJump is based on the principle that, if we rate-limit all hosts so that they can only issue one packet every network epoch, then no packet will take more than one network epoch to be delivered to the destination even in the worst case.

Latency Variance vs. Throughput

Although the equation derived above provides an absolute upper bound on in-network delay, it also aggressively restricts throughput. Formulating Equation 2 for throughput, we obtain:

$$throughput = \frac{P}{network\ epoch} = \frac{R}{2n} \qquad (3)$$

For example, with 1,000 hosts and a 10 Gb/s edge, we obtain an effective throughput of 5 Mb/s per host. Clearly, this is not ideal. We can improve this situation by making two observations. First, Equation 2 is pessimistic: it assumes that all hosts transmit to one destination at the worst time, which is unlikely given a realistic network and traffic distribution. Second, some applications, like PTPd, are more sensitive to interference than others—for example, memcached and Naiad—whereas still other applications, like Hadoop, are more sensitive to throughput restrictions. From the first observation, we can relax the throughput constraints in Equation 2 by assuming that fewer than n hosts send to a single destination at the worst time. For example, if we guess that only 500 of the 1,000 hosts concur-

rently send to a single destination, then those 500 hosts can send at twice the rate and maintain the same network delay if our assumption holds. More generally, we define a scaling factor f so that the assumed number of senders n' is given by:

$$n' = \frac{n}{f} \quad \text{where} \quad 1 \leq f \leq n. \qquad (4)$$

Intuitively, f is a "throughput factor": as the value of f grows, so does the available bandwidth.

From the second observation, some (but not all) applications can tolerate some degree of latency variance. Instead, for these applications we aim for a statistical reduction in latency variance. This reintroduces a degree of statistical multiplexing to the network, albeit one that is more tightly controlled. When the guess for f is too optimistic (the actual number of senders is greater than n'), some queueing occurs, causing interference.

The probability that interference occurs increases with increasing values of f. At the upper bound ($f = n$), latency variance is similar to existing networks and full network throughput is available. At the lower bound ($f = 1$), latency is guaranteed, albeit with reduced throughput. In essence, f quantifies the latency variance vs. throughput tradeoff.

Jump the Queue with Prioritization

We would like to use multiple values of f concurrently, so that different applications can benefit from the latency variance vs. throughput tradeoff that suits them best. To achieve this, we partition the network so that traffic from latency-sensitive applications, like PTPd, memcached, and Naiad can "jump-the-queue" over traffic from throughput-intensive applications like Hadoop. Ethernet switches support the IEEE 802.1Q standard, which provides eight (0–7) hardware enforced "service classes" or "priorities."

The problem with using priorities is that they can become a "race to the top." For example, memcached developers may assume that memcached traffic is the most important and should receive the highest priority to minimize latency. Meanwhile, Hadoop developers may assume that Hadoop traffic is the most important and should similarly receive the highest priority to maximize throughput. Since there are a limited number of priorities, neither can achieve an advantage and prioritization loses its value. QJump is different: it intentionally binds priority values to rate-limits. High priorities are given aggressive rate limits (small f values), and priorities thus become useful because they are no longer "free." QJump users must choose between low latency variance at low throughput (high priority) and high latency variance at high throughput (low priority). We call the assignment of an f value to a priority a "QJump level." The latency variance of a given QJump level depends on the number of QJump levels above it and their traffic patterns.

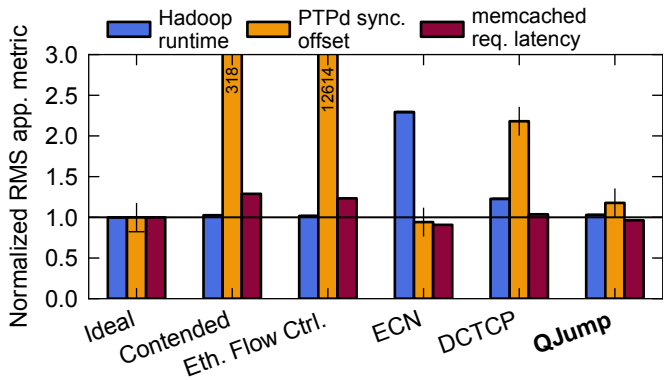


Figure 3: Comparison of QJump to several available congestion control alternatives

Implementation

QJump has two components: a rate-limiter to provide admission control to the network, and an application utility to configure unmodified applications to use QJump levels. Our full paper describes the rate limiter and application utility in detail, and the source code for both is available from our Web site.

In our prototype, we use our own high-performance rate limiter built upon the queueing discipline (qdisc) mechanism offered by the Linux kernel traffic control (TC). TC modules do not require kernel modifications and can be inserted and removed at runtime, making them flexible and easy to deploy.

To support unmodified applications, we implemented a utility that dynamically intercepts socket setup system calls and alters their options. We inject the utility into unmodified executables via the Linux dynamic linker's `LD_PRELOAD` support.

Performance Comparison

We have already demonstrated that QJump can resolve network interference, but how does it compare to existing congestion control mechanisms? To find out, we have tested QJump against several readily deployable congestion control schemes. In these experiments, PTPd, memcached, and Hadoop are configured to run on the same network for a 10-minute period. Since interference is transient in these experiments, we measure the degree to which it affects applications using the root mean square (RMS) of each application-specific metric. For Hadoop, the metric of interest is the job runtime, for PTPd it is the time synchronization offset, and for memcached it is the request latency. Figure 3 shows six cases: an ideal case, a contended case, and one for each of the four comparison schemes. All cases are normalized to the ideal case, which has each application running alone on an idle network.

Ethernet Flow Control

Like QJump, Ethernet Flow Control is a data link layer congestion control mechanism. Hosts and switches issue special *pause* messages when their queues are nearly full, alerting senders to slow down. Figure 3 shows that Ethernet Flow Control (Pause frames) has a limited positive impact on memcached but increases the RMS offset for PTPd. Hadoop's performance remains unaffected.

Early Congestion Notification (ECN)

ECN is a network-layer mechanism in which switches indicate queueing to end hosts by marking TCP packets. Our Arista 7050 switches implement ECN with weighted random early detection (WRED). The effectiveness of WRED depends on an administrator correctly configuring upper and lower marking thresholds. We investigated 10 different marking threshold pairs, ranging between [5, 10] and [2560, 5120], in packets. None of these settings achieved ideal performance for all three applications, but the best compromise was [40, 80]. With this configuration, ECN very effectively resolves the interference experienced by PTPd and memcached. However, this comes at the expense of increased Hadoop job runtimes.

Datacenter TCP (DCTCP)

DCTCP uses the rate at which ECN markings are received to build an estimate of network congestion. It applies this to a new TCP congestion avoidance algorithm to achieve lower queueing delays. We configured DCTCP with the recommended ECN marking thresholds of [65, 65]. Figure 3 shows that DCTCP reduces the variance in PTPd synchronization and memcached latency compared to the contended case. However, this comes at an increase in Hadoop job runtimes, as Hadoop's bulk data transfers are affected by DCTCP's congestion avoidance.

QJump

Figure 3 shows that QJump achieves the best results. The variance in Hadoop, PTPd, and memcached performance is close to the uncontended ideal case.

Conclusion

QJump applies QoS-inspired concepts to datacenter applications to mitigate network interference. It offers multiple QJump levels with different latency variance vs. throughput tradeoffs, including bounded latency (at low rate) and full utilization (at high latency variance). QJump is readily deployable, open source, and requires no hardware, protocol, or application changes.

Our source code and all experimental data sets are available at http://www.cl.cam.ac.uk/research/srg/netos/qjump.

Acknowledgments

The full paper version of this work includes contributions from Jon Crowcroft, Steven Hand, and Robert N. M. Watson. This work was jointly supported by a Google Fellowship, EPSRC INTERNET Project EP/H040536/1, the Defense Advanced Research Projects Agency (DARPA), and the Air Force Research Laboratory (AFRL), under contract FA8750-11-C-0249. The views, opinions, and/or findings contained in this article are those of the authors and should not be interpreted as representing the official views or policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the Department of Defense.

References

[1] J. Crowcroft, S. Hand, R. Mortier, T. Roscoe, and A. Warfield, "QoS's Downfall: At the Bottom, or Not at All!" in *Proceedings of the ACM SIGCOMM Workshop on Revisiting IP QoS,* 2003, pp. 109–114.

[2] J. Dean and L. A. Barroso, "The Tail at Scale: Managing Latency Variability in Large-Scale Online Services," *Communications of the ACM*, vol. 56, no. 2 (Feb. 2013), pp. 74–80.

[3] M. P. Grosvenor, M. Schwarzkopf, I. Gog, R. N. M. Watson, A. W. Moore, S. Hand, and J. Crowcroft, "Queues Don't Matter if You Can JUMP Them!" forthcoming in *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI '15),* May 2015.

[4] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, and P. Barham, "Naiad: A Timely Dataflow System," in *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP),* 2013, pp. 439–455.

[5] N. G. Duffield, P. Goyal, A. Greenberg, P. Mishra, K. K. Ramakrishnan, and J. E. van der Merive, "A Flexible Model for Resource Management in Virtual Private Networks," in *Proceedings of the ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM),* Aug. 1999, pp. 95–108.

usenix
THE ADVANCED
COMPUTING SYSTEMS
ASSOCIATION

USENIX Awards

USENIX honors members of the community with three prestigious annual awards which recognize public service and technical excellence. The winners of these awards are selected by the USENIX Awards Committee. The USENIX membership may submit nominations for any or all three of the awards to the committee.

The USENIX Lifetime Achievement (Flame) Award

The USENIX Lifetime Achievement Award recognizes and celebrates singular contributions to the UNIX community in both intellectual achievement and service that are not recognized in any other forum. The award itself is in the form of an original glass sculpture called "The Flame," and in the case of a team based at a single place, a plaque for the team office.

Details and a list of past recipients are available at www.usenix.org/about/flame.

The Software Tools Users Group (STUG) Award

The Software Tools Users Group Award recognizes significant contributions to the general community that reflect the spirit and character of those who came together to form the Software Tools Users Group (STUG). This is a cash award.

STUG and the Software Tools effort were characterized by two important tenets. The first was an extraordinary focus on building portable, reusable libraries of code shared among multiple applications on wildly disparate systems. The other tenet, shared with the UNIX community, is "renegade empowerment."

The Software Tools Users Group gave users the power to improve their environment when their platform provider proved inadequate, even when local management sided with the platform provider. Therefore, nominees for the STUG Award should exhibit one or both of these traits in a conspicuous manner: a contribution to the reusable code-base available to all or the provision of a significant enabling technology directly to users in a widely available form.

Details and a list of past recipients are available at www.usenix.org/about/stug.

The LISA Award for Outstanding Achievement in System Administration

This annual award goes to someone whose professional contributions to the system administration community over a number of years merit special recognition.

Details and a list of past recipients are available at www.usenix.org/lisa/awards/outstanding.

NOMINATIONS DUE MAY 1

Call for Award Nominations

USENIX requests nominations for these three awards; they may be from any member of the community. Nominations should be sent to the Chair of the Awards Committee via awards@usenix.org by May 1 each year. A nomination should include:

1. Name and contact information of the person making the nomination
2. Name(s) and contact information of the nominee(s)
3. A citation, approximately 100 words long
4. A statement, at most one page long, on why the candidate(s) should receive the award
5. Between two and four supporting letters, no longer than one page each

www.usenix.org/about/usenix-awards

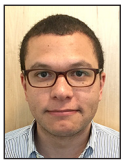
The Design and Implementation of Open vSwitch

BEN PFAFF, JUSTIN PETTIT, TEEMU KOPONEN, ETHAN J. JACKSON, ANDY ZHOU, JARNO RAJAHALME, JESSE GROSS, ALEX WANG, JONATHAN STRINGER, PRAVIN SHELAR, KEITH AMIDON, AND MARTIN CASADO

Ben Pfaff is a Lead Developer of the Open vSwitch project. He was a founding employee at Nicira and is currently at VMware. He received his PhD from Stanford University in 2007. blp@cs.stanford.edu

Justin Pettit is a Lead Developer on the Open vSwitch project. He was a founding employee at Nicira and previously worked at three successful startups focused on network security. He received his master's degree in computer science at Stanford University. jpettit@cs.stanford.edu

Teemu Koponen was the Chief Architect at Nicira before joining VMware. Teemu received his PhD from Helsinki University of Technology in 2008 and ever since has been indecisive enough to remain active within the network research community while working for the industry. He received the ACM SIGCOMM Rising Star Award 2012 for his contributions on network architectures. tkoponen@vmware.com



Ethan Jackson is a Staff Engineer at VMware and a researcher at UC Berkeley. His primary focus is on SDN, Network Function Virtualization, and high performance software switching. ejj@ej2.org

Before joining the Open vSwitch team, Andy Zhou worked on many networking and network security products using multicore NPUs. His other interests include embedded system, kernel, and computer architectures. He received his MSCS from Carnegie Mellon University. azhou@nicira.com

Open vSwitch is the most widely used virtual switch in cloud environments. Open vSwitch is a multi-layer, open source virtual switch for all major hypervisor platforms. It was designed de novo for networking in virtual environments, resulting in major design departures from traditional soft switching architectures. We detail the advanced flow classification and caching techniques that Open vSwitch uses to optimize its operations and conserve hypervisor resources. These implementation details will benefit anyone who uses Open vSwitch.

Virtualization has changed the way we do computing over the past 15 years; for instance, many datacenters are entirely virtualized to provide quick provisioning, spillover to the cloud, and improved availability during periods of disaster recovery. While virtualization has yet to reach all types of workloads, the number of virtual machines has already exceeded the number of servers and shows no signs of stopping [1].

The rise of server virtualization has brought with it a fundamental shift in datacenter networking. A new network access layer has emerged in which most network ports are virtual, not physical [2], and the first hop switch for workloads, therefore, increasingly resides within the hypervisor. In the early days, these hypervisor "vswitches" were primarily concerned with providing basic network connectivity. In effect, they simply mimicked their ToR (top-of-rack) cousins by extending physical L2 networks to resident virtual machines. As virtualized workloads proliferated, the limits of this approach became evident: reconfiguring and preparing a physical network for new workloads slows their provisioning, and coupling workloads with physical L2 segments severely limits their mobility and scalability to that of the underlying network.

These pressures resulted in the emergence of network virtualization [3]. In network virtualization, virtual switches become the primary provider of network services for VMs, leaving physical datacenter networks with transportation of IP tunneled packets between hypervisors. This approach allows the virtual networks to be decoupled from their underlying physical networks, and by leveraging the flexibility of general purpose processors, virtual switches can provide VMs, their tenants, and administrators with logical network abstractions, services, and tools identical to dedicated physical networks.

Network virtualization demands a capable virtual switch—forwarding functionality must be wired on a per-virtual-port basis to match logical network abstractions configured by administrators. Implementation of these abstractions, across hypervisors, also greatly benefits from fine-grained centralized coordination. This approach starkly contrasts with early virtual switches for which static, mostly hardcoded forwarding pipelines had been completely sufficient to provide virtual machines with L2 connectivity to physical networks.

It was this context—the increasing complexity of virtual networking, emergence of network virtualization, and the limitations of existing virtual switches—that allowed Open vSwitch to quickly gain popularity. Today, on Linux, its original platform, Open vSwitch works with most hypervisors and container systems, including Xen, KVM, and Docker. Open vSwitch

Jarno Rajahalme is part of the Open vSwitch team at VMware and has specialized in the OVS flow classifier algorithms. He received his doctor of science in technology degree from Aalto University in 2012, and is the author or co-author of tens of patents and several conference and journal papers. jrajahalme@nicira.com

Jesse Gross works on the Open vSwitch team at VMware where he has led the development of several protocols used for network virtualization. Jesse was also the original maintainer of the kernel components of Open vSwitch in Linux. He holds a degree in computer science from Stanford. jgross@vmware.com

Alex Wang is a developer on Open vSwitch. He received his master's degree in electrical engineering from UC San Diego. ee07b291@gmail.com

Jonathan Stringer hails from New Zealand, where he studied computer science specializing in networks. He's previously been involved in SDN deployments in New Zealand and now actively works on the Open vSwitch team at VMware. joe@wand.net.nz

Pravin Shelar is an Open vSwitch developer. He is currently the OVS kernel module maintainer. His most recent focus has been on tunneling. pshelar@nicira.com

Keith Amidon has spent 20+ years building high performance networks and networking software for forwarding and security. He managed the Open vSwitch development team at Nicira/VMware and recently co-founded a stealth-mode network security startup. keith@awakenetworks.com

Martin Casado is a Fellow and the SVP and GM of the Networking & Security Business Unit at VMware. He was the co-founder and CTO of Nicira Networks. He received his PhD from Stanford University where he remains a Consulting Assistant Professor. mcasado@vmware.com

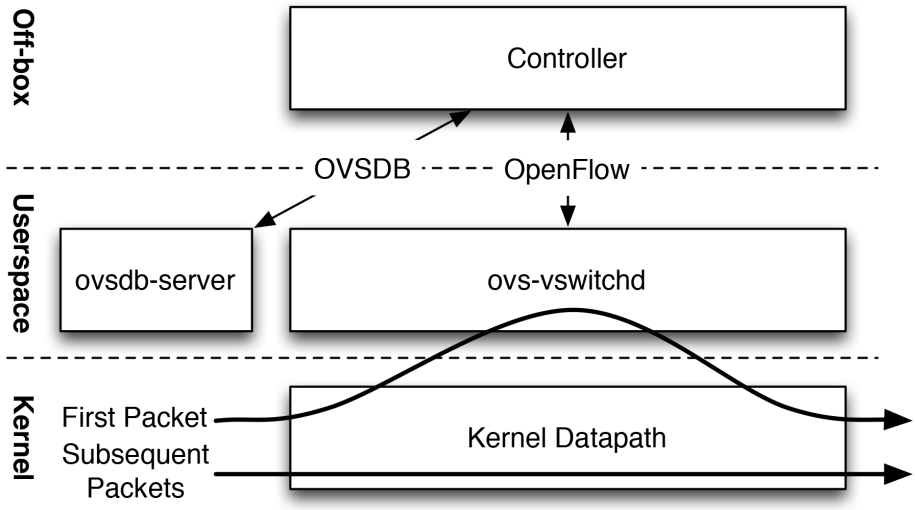


Figure 1: The components and interfaces of Open vSwitch. The first packet of a flow results in a miss, and the kernel module directs the packet to the userspace component, which caches the forwarding decision for subsequent packets into the kernel.

also works "out of the box" on the FreeBSD and NetBSD operating systems, and ports to the VMware ESXi and Microsoft Hyper-V hypervisors are underway.

In this article, we give a brief overview of the design and implementation of Open vSwitch [4]. The key elements of its design revolve around the performance required by the production environments in which Open vSwitch is commonly deployed, and the programmability demanded by network virtualization. Unlike traditional network appliances, whether software or hardware, which achieve high performance through specialization, Open vSwitch is designed for flexibility and general-purpose use. It must achieve high performance without the luxury of specialization, adapting to differences in platforms supported, all while sharing resources with the hypervisor and its workloads. For a more complete description of Open vSwitch and its performance evaluation, see our upcoming paper in the proceedings of the USENIX NSDI '15 conference [6].

Design

In Open vSwitch, two major components direct packet forwarding. The first, and larger, component is `ovs-vswitchd`, a userspace daemon that is essentially the same from one operating system and operating environment to another. The other major component, a *datapath kernel module*, is usually written specially for the host operating system for performance.

Figure 1 depicts how the two main OVS components work together to forward packets. It is the kernel datapath module that receives the packets first, from a physical NIC or a VM's virtual NIC. There are then two possibilities: either `ovs-vswitchd` has given the datapath instructions on how to handle packets of this type or it has not. In the former case, the datapath module simply follows the instructions, called *actions*, which list physical ports or tunnels on which to transmit the packet. Actions may also specify packet modifications, packet sampling, or instructions to drop the packet. In the other case, where the datapath has not been told what to do with the packet, it delivers it to `ovs-vswitchd`. In userspace, `ovs-vswitchd` determines how the packet should be handled, then it passes the packet back to the datapath with the desired handling. Usually, `ovs-vswitchd` also tells the datapath to cache the actions, for handling similar future packets.

Open vSwitch is commonly used as an SDN switch, and the main way to control forwarding is OpenFlow [5]. It is the responsibility of `ovs-vswitchd` to receive OpenFlow flow tables from an SDN controller, match any packets received from the datapath module against these OpenFlow tables, gather the actions applied, and finally cache the result in the kernel datapath. This allows the datapath module to remain unaware of the particulars of the OpenFlow wire protocol, further simplifying it. From the OpenFlow controller's point of view, the caching and separation into user and kernel components are invisible implementation details; in the controller's view, each packet visits a series of OpenFlow flow tables, and the switch finds the highest-priority flow whose conditions are satisfied by the packet and executes its OpenFlow actions.

Flow Cache Design

Algorithmic packet classification is expensive on general purpose processors, and packet classification in the context of OpenFlow is especially costly because of the generality of the form of the match, which may test any combination of Ethernet addresses, IPv4 and IPv6 addresses, TCP and UDP ports, and many other fields, including packet metadata such as the switch ingress port. This cost is amplified by the large number of flow tables used by sophisticated SDN controllers: for example, VMware NSX [3] uses about 15 classifications per packet at minimum.

Open vSwitch uses two strategies to maximize performance in the face of expensive packet classification. The first strategy is to optimize the classification itself, by refining the classification algorithms and our implementations of them over time. The second strategy is to perform fewer classifications through effective use of caching. This section introduces the flow cache design, and the following section delves into the details.

Open vSwitch's kernel datapath initially cached *microflows*, that is, each cache entry had to match on all of the fields supported by OpenFlow. Microflow caching is very fine-grained: each cache entry matches, roughly, one stream of packets in a single transport connection. A microflow cache can be implemented as a hash table, which allows the kernel module to be very simple.

Microflow caching is effective with the most common network traffic patterns, but it seriously degrades when faced with large numbers of short-lived connections. In such cases, many packets miss the cache and must not only cross the kernel-userspace boundary, but also execute a long series of expensive packet classifications. In production, this kind of traffic can be caused by port scans, network management tools, P2P applications, malware, and other sources. None of these is common, but they happen often enough that customers notice the issue.

To improve performance under those traffic patterns, we augmented the microflow cache with a *megaflow cache*. The megaflow cache is a single flow lookup table that supports generic matching, i.e., it supports caching forwarding decisions for larger aggregates of traffic than connections through wildcarding. The megaflow cache somewhat resembles a general-purpose OpenFlow table, but it is simpler in two ways: it does not have priorities, which speeds up packet classification because any match is a "best match," and there is only one megaflow table, instead of a pipeline of them, so any packet needs only one classification rather than a series. In the common case, a megaflow lookup remains more expensive than a microflow cache lookup, so we retained the microflow cache as a first-level "exact-match cache," consulted before the megaflow cache. This reduces the cost of megaflows from per-packet to per-microflow.

Caching-Aware Packet Classification

Open vSwitch uses a *tuple space search* classifier [7] for all of its packet classifications, both kernel and userspace. To understand how tuple space search works, imagine that all the flows in an Open vSwitch flow table matched on the same fields in the same way: for example, all flows match the source and destination Ethernet address but no other fields. A tuple search classifier implements such a flow table as a single hash table. If the controller then adds new flows with a different form of match, the classifier creates a second hash table that hashes on the fields matched in those flows. With two hash tables, a search must look in both hash tables. If there are no matches, the flow table doesn't contain a match; if there is a match in one hash table, that flow is the result; if there is a match in both, then the result is the flow with the higher priority. As the controller continues to add more flows with new forms of match, the classifier similarly expands to include a hash table for each unique match, and a search of the classifier must look in every hash table.

As Open vSwitch userspace processes a packet through its OpenFlow tables, it tracks the packet field bits that were consulted as part of the forwarding decision. This technique constructs an effective megaflow cache from simple OpenFlow flow tables. For example, if the OpenFlow table only looks at Ethernet addresses (as would a flow table based on L2 MAC learning), then its megaflows will also look only at Ethernet addresses. On the other hand, if even one flow entry in the table matches on the TCP destination port, tuple space search examines TCP destination port of every packet, so that every packet in, for example, a port scan must go to userspace, and performance drops.

However, in the latter case, a more sophisticated classifier may be able to notice cases where the match on TCP destination can be omitted. Thus, after introduction of megaflows, much of our performance work on Open vSwitch has centered around making userspace generate megaflows that match on fewer fields. The following sections describe improvements of this type that we have integrated into Open vSwitch.

Tuple Priority Sorting

Lookup in a tuple space search classifier ordinarily requires searching every tuple. Even if a search of an early tuple finds a match, the search must still look in the other tuples because one of them might contain a matching flow with a higher priority. We improved on this by searching the hash tables from largest to smallest maximum priority. Then a successful search can often terminate early because the current match is known to be higher-priority than any possible later match.

Staged Lookup

Even if a tuple includes many fields, a single field might be enough to tell that a search must fail: for example, if all the flows match on a destination IP that is different from the one in the packet we are looking up, then it suffices to just examine the destination IP field. The staged lookup optimization makes use of this observation by adding to a generated megaflow only match fields actually needed to determine that the tuple's flows did not match.

The tuple implementation as a hash table over all its fields made such an optimization difficult. One cannot search a hash table on a subset of its key. We considered other data structures, such as tries or per-field hash tables, but these increased search time or space requirements unacceptably.

The solution we implemented statically divides fields into four groups, in decreasing order of traffic granularity: metadata (e.g., the switch ingress port), L2, L3, and L4. We changed each tuple from a single hash table to an array of four hash tables, called *stages*: one over metadata fields only; one over metadata and L2 fields; one over metadata, L2, and L3 fields; and one over all fields. A lookup in a tuple searches each of its stages in order. If any search turns up no match, then the overall search of the tuple also fails, and only the fields included in the stage last searched must be added to the megaflow match.

Prefix Tracking

Flows in OpenFlow often match IPv4 and IPv6 subnets to implement routing. When all the flows that match on such a field use the same subnet size, for example, all match /16 subnets, this works out fine for constructing megaflows. If, on the other hand, different flows match different subnet sizes, like any standard routing table does, the constructed megaflows match the longest subnet prefix: for example, any host route (/32) forces all the megaflows to match full addresses. Suppose, for example, Open vSwitch is constructing a megaflow for a packet addressed to 10.5.6.7. If flows match subnet 10/8 and host 10.1.2.3/32, one could safely install a megaflow for 10.5/16 (because 10.5/16 is completely inside 10/8 and does not include 10.1.2.3), but without additional optimization Open vSwitch installs 10.5.6.7/32.

We implemented optimization of prefixes for IPv4 and IPv6 fields using a trie structure. If a flow table matches over an IP address, the classifier executes an LPM lookup for any such field *before* the tuple space search, both to determine the maximum megaflow prefix length required, as well as to determine which tuples can be skipped entirely without affecting correctness.

We also adopted prefix tracking for L4 transport port numbers. This prevents high-priority ACLs that match specific ports from forcing all megaflows to match the entire port field.

Cache Invalidation

The flip side of caching is the complexity of managing the cache. Ideally, Open vSwitch could precisely identify the megaflows that need to change in response to some event. For some kinds of events, this is straightforward, but the generality of the OpenFlow model makes precise identification difficult in other cases. One example is adding a new flow to an OpenFlow table. Any megaflow that matches a flow in that OpenFlow table whose priority is less than the new flow's priority should potentially now exhibit different behavior, but we do not know how to efficiently (in time and space) identify precisely those flows. The problem is worsened by long sequences of OpenFlow flow table lookups. We concluded that precision is not practical in the general case.

To revalidate the cached flows, Open vSwitch has to examine every datapath flow for possible changes. Each flow has to be passed through the OpenFlow flow table in the same way as it was originally constructed so that the generated actions can be compared against the ones currently installed in the datapath. This is time-consuming if there are many datapath flows or if the OpenFlow flow tables are complicated. Older versions of Open vSwitch were single-threaded, which meant that the time spent reexamining all of the datapath flows blocked setting up new flows for arriving packets that did not match any existing datapath flow. This added high latency to flow setup for those packets, greatly increased the overall variability of flow setup latency, and limited the overall flow setup rate. Therefore, Open vSwitch had to limit the maximum number of cached flows installed in the datapath to around 1,000. When Open vSwitch 2.1 introduced multiple dedicated threads for cache revalidation, we were able to scale the revalidation performance to match the flow setup performance, as well as greatly increase the maximum kernel cache size, to about 200,000 entries.

Open vSwitch userspace obtains datapath cache statistics by periodically (about once per second) polling the kernel module for every flow's packet and byte counters. The core use of datapath flow statistics is to determine which datapath flows are useful and should remain installed in the kernel and which ones are not processing a significant number of packets and should be evicted. Short of the table's maximum size, flows remain in the datapath until they have been idle for a configurable amount

of time, which now defaults to 10 seconds. Above the maximum size, Open vSwitch drops this idle time to force the table to shrink. The threads that periodically poll the kernel for per-flow statistics also use those statistics to implement OpenFlow's per-flow packet and byte count statistics and flow idle timeout features. This means that OpenFlow statistics are themselves only periodically updated.

The above description covers the invalidation strategy of the megaflow cache. The invalidation of the first-level microflow cache (discussed in the Flow Cache Design section) is much simpler. The kernel only opportunistically invalidates microflow entries: when a microflow cache results in a miss and the megaflow cache is about to insert a new microflow entry, an existing microflow entry is replaced if the entry hashes to a hash table bucket already in use.

Conclusion

We described the design and implementation of Open vSwitch, an open source, multi-platform OpenFlow virtual switch. Open vSwitch has simple origins, but its performance has been gradually optimized to match the requirements of multi-tenant datacenter workloads, which has necessitated a more complex design. Given its operating environment, we anticipate no change of course but expect its design only to become more distinct from traditional network appliances over time.

References

[1] T. J. Bittman, G. J. Weiss, M. A. Margevicius, and P. Dawson, "Magic Quadrant for x86 Server Virtualization Infrastructure," Gartner, June 2013.

[2] Crehan Research Inc. and VMware Estimate, March 2013.

[3] T. Koponen, K. Amidon, P. Balland, M. Casado, A. Chanda, B. Fulton, I. Ganichev, J. Gross, P. Ingram, E. Jackson, A. Lambeth, R. Lenglet, S.-H. Li, A. Padmanabhan, J. Pettit, B. Pfaff, R. Ramanathan, S. Shenker, A. Shieh, J. Stribling, P. Thakkar, D. Wendlandt, A. Yip, and R. Zhang, "Network Virtualization in Multi-Tenant Datacenters," in *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI '14)*, Seattle, WA, April 2014.

[4] Open vSwitch: http://www.openvswitch.org, September 2014.

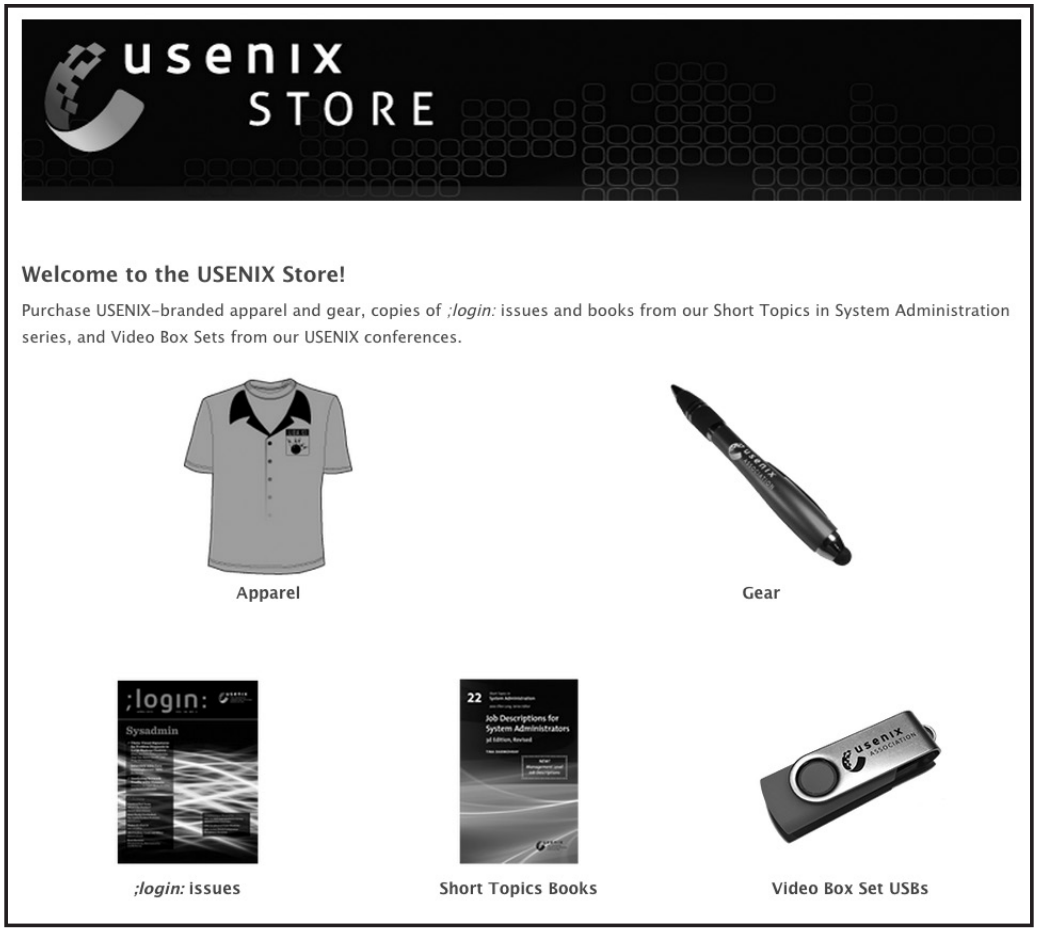
[5] OpenFlow: https://www.opennetworking.org/sdn-resources/openflow, January 2014.

[6] B. Pfaff, J. Pettit, T. Koponen, E. J. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, K. Amidon, and M. Casado, "The Design and Implementation of Open vSwitch," in *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI '15)*, Oakland, CA, May 2015.

[7] V. Srinivasan, S. Suri, and G. Varghese, "Packet Classification Using Tuple Space Search," in *Proceedings of the ACM SIGCOMM '99 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, 1999.

usenix
THE ADVANCED
COMPUTING SYSTEMS
ASSOCIATION

The USENIX Store is Open for Business!



Want to buy a subscription to *;login:*, the latest short topics book, a USENIX or conference shirt, or the box set from last year's workshop? Now you can, via the brand new USENIX Store!

Head over to www.usenix.org/store and check out the collection of t-shirts, video box sets, *;login:* magazines, short topics books, and other USENIX and LISA gear. USENIX and LISA SIG members save, so make sure your membership is up to date.

www.usenix.org/store

Interview with Andrew Tanenbaum

RIK FARROW

Andrew S. Tanenbaum was born in New York City and raised in White Plains, NY. He has a BS from MIT and a PhD from the University of California at Berkeley. He is currently a Professor of Computer Science at the Vrije Universiteit in Amsterdam. Professor Tanenbaum is the principal designer of three operating systems: TSS-11, Amoeba, and MINIX. In addition, he is the author or coauthor of five books, which together have been translated into more than 20 languages. In 2004 Tanenbaum became an Academy Professor, which carried with it a five-year grant totaling one million euro to do research on reliable operating systems. His university matched this amount. In 2008 he received a prestigious European Research Council grant of 2.5 million euro to continue this research. Tanenbaum is a Fellow of the ACM, a Fellow of the IEEE, and a member of the Netherlands Royal Academy of Arts and Sciences. In 1994 he was the recipient of the ACM Karl V. Karlstrom Outstanding Educator Award. In 1997 he won the ACM SIGCSE Award for Outstanding Contribution to Computer Science Education. In 2007 he won the IEEE James H. Mulligan, Jr. Education Medal. ast@cs.vu.nl

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

Although I had certainly encountered Andrew Tanenbaum at other conferences, the first time I remember talking to him was in a "terminal" room at USENIX Annual Tech in 2004. By that time, a terminal room was a place where you could have a hardwired connection to the Internet, and Andy showed me www.electoral-vote.com, a political Web site which he was working on that analyzes polling data. For someone who focuses on working with PhD students, and writing books and operating systems, building such a useful political site seemed a bit far afield to me. But the more you know about Andy, the more you learn just how broad his interests are.

After that meeting, we would usually spend some time talking at the systems conferences that we both attended. While we mostly discussed MINIX 3 (www.minix3.org), we also talked about other things, such as his current position within Vrije Universiteit in The Netherlands. If you are wondering how Andy wound up there, you should read his FAQ [1]. But given the brief meetings, there were some things I didn't get to ask him, and with his past experience in distributed systems, I thought that now would be a good time to interview him.

Rik: While at the OSDI conference (2014), I heard someone mention that people have forgotten about all the work that was done on building distributed or parallel systems in the 1980s and early '90s. Could you explain why there was such strong interest in systems like Amoeba [2, 3], and Sprite?

Andy: There wasn't a lot of commercial interest in parallel or distributed systems in the 1980s, but there was some in academia from people who try to stay ahead of the curve. Already then, cheap workstations and PCs existed, and it occurred to some people that you could harness them together and get a bigger bang for the buck than buying a supercomputer. At the University of Wisconsin, for example, there was work on harvesting the power of idle workstations to form an ad hoc supercomputer.

My work consisted of putting sixteen Motorola 68000s in a rack and letting people start jobs there from their desktop machines without having to worry too much about the details. We called the rack the "processor pool" and built an operating system (Amoeba) to control it. We published some papers about it, but it didn't get much attention in commercial circles. Nowadays it is called "cloud computing" and gets a lot of attention.

Rik: I think people often forget just how slow processors were in the '80s, right into the early '90s. My first UNIX system, a 68010, had a blistering clock rate of 10 MHz (1983). A 1987 Sun-4/260 ran at 16.67 MHz, and was noticeably faster than the 68030s it replaced. Having a rack of systems, where a user could run programs on the least busy one, surely must have seemed like a great idea.

Andy: What the Amoeba processor pool did was create a shared resource, which is more efficient than dedicated ones. If all the money available for computing resources was spent to give each user the most powerful computer you could buy for that amount, you would often have the situation that one user needed a lot of computing power for a short time, for example, to run "make" to compile a big program, while all the other computers were idle but unavailable to the one who needed the power. By having a processor pool available to everyone, if one user needed the whole thing and nobody else needed any computing, the one user could get all of it. If two users needed a lot of computing, they would each get half. So the model was to put most of the power in the processor pool and just give users simple terminals. This whole model foreshadowed cloud computing, which also centralizes the computing power and lets you take as much as you need for a short period and nothing when you are sitting around scratching your head deciding what to do next.

Rik: Did the work on Amoeba have anything to do with the development of MINIX, the operating system you wrote to help students learn about operating systems?

Andy: Amoeba didn't influence the development of MINIX much as Amoeba was intended as a research vehicle and not as a UNIX clone. For example, the Amoeba file system, the bullet server, wrote files onto the disk as consecutive sectors so they could be read back at very high speed (basically one disk command to read a whole file). Files were also immutable. The system was based on cryptographically secure capabilities managed directly by user programs. It was an attempt to push the envelope on research and was completely different from MINIX, which was initially intended for teaching students how a UNIX-like system worked inside.

Rik: MINIX started out as a microkernel, moving away from the generally accepted design of monolithic kernels, which are still dominant today. What were the advantages of using a microkernel for MINIX?

Andy: Since my initial goal in writing MINIX was to have students learn from it, I thought that breaking it into a number of smaller chunks that interacted in very well defined ways would make it easier to understand. Generally speaking, for example, six programs of 2000 lines are easier to understand than one program of 12,000 lines.

But also from the beginning, I was aware that putting most of the operating system in "user mode" as multiple processes would make it more reliable and more secure against attempts to hack it. Now the 8088 didn't have kernel and user modes, but I assumed that some future version of the 8088 would have them, and that is what happened, of course.

Rik: So why don't we see more microkernels used today?

Andy: Because they are mostly used in embedded systems, where reliability matters. In mission-critical embedded systems, microkernels like QNX are widely used but they are invisible to the user. Also, L4 is used in the radio chip inside over a billion smartphones. I think monolithic kernels are mostly used due to inertia, whereas for each new embedded system the designers look around and see what is best right now without worrying too much about legacy. Performance used to be a problem with microkernels, but L4 showed this is not inherent. In many other areas legacy systems dominate, even though they are inferior to other ones.

For example, I have never heard an argument why the furlong-stone-fortnight system used in the UK and US is better than the metric system other than "We've always done it that way." Consider Fahrenheit vs. Celsius. Try arguing that the NTSC (Never Twice the Same Color) television system is better than the alternatives. What about point-and-shoot cameras that have an aspect ratio of 4:3, like 1950s TV sets? C is still widely used even though it is not type safe, and C programs are subject to buffer overflow attacks and more. COBOL is horrible but lasted for decades. In general, once some technology gets established, it is very hard to dislodge.

I think the research community is too fixated on Linux, and any monoculture is bad. Even a stable, mature, open-source system like FreeBSD hardly gets any attention.

Rik: I've written many times that running microkernels on current CPU architectures cannot work well, as microkernels and monolithic kernels rely on very different designs for system communication. Monolithic kernels keep all modules in one privileged address space, which is convenient, fast, as well as considerably less secure. Microkernels minimize the amount of code running within the privileged address space, but at the cost of having to make context changes when communicating with or between system modules. Also, unprivileged modules need privileged access for many of the tasks they perform.

Do system architecture changes like the IOMMU [4], as well as others I either don't know about or haven't imagined yet, help microkernels run as fast or faster than monolithic ones, but with a much higher level of security?

Andy: Better hardware certainly helps but I don't think IOMMUs are necessarily the answer. One thing that may help is multicore architectures. One of my PhD students has been doing research on a prototype system in which the major server components each run on their own core. This way when it is needed, there is no context switching, the cache is warm, and the server is ready to run with no overhead. As we move toward a world in which all chips

have cores to spare, this could make microkernels more competitive since people won't worry about "wasting" cores, just as no one worries about "wasting" RAM on bloated software now.

Rik: Now that you've been retired [5] from Vrije Universiteit, what do you plan on doing? You've always stayed busy, much more so than most people.

Andy: For one thing, I will continue teaching one course I give in our masters program (on how to write a grant proposal). I also still have five PhD students to supervise.

For another, I want to continue publicizing MINIX 3. It is more popular than many people realize. According to the statistics from the log, visible at minix3.org/stats, we had over 60,000 downloads of the .iso file in 2014 and over 600,000 since 2007. The minix3.org site has had over 3 million visits since I put the counter on there about five years ago.

Still, I would like to build a more active community. One thing I will probably do in that respect is sign up for the ACM Distinguished Speakers Program and give lectures about MINIX at universities. I need to maintain my Platinum Medallion status on Delta Airlines somehow :-)

Furthermore, I have five books that are current and in constant need of new editions. Fortunately, I have excellent coauthors to help me out.

In addition, I had about 50,000 of my negatives and slides scanned in, and I want to organize, label, and clean them up with Photoshop. I also have a couple hundred hours of video that need work. I recently bought a Mac Pro (garbage can model) to handle the video processing.

So I don't think I'll be bored, for a few months, anyway.

References

[1] Andrew S. Tanenbaum's FAQ: http://www.cs.vu.nl/~ast/home/faq.html.

[2] A. Tanenbaum et al, "The Amoeba Distributed Operating System": http://www.cs.vu.nl/pub/amoeba/Intro.pdf.

[3] This page has links to papers, as well as a good description of Amoeba: http://en.wikipedia.org/wiki/Amoeba_%28operating_system%29.

[4] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe, "Arrakis: The Operating System Is the Control Plane," OSDI '14.

[5] Retirement: http://www.few.vu.nl/~ast/afscheid/.

usenix
THE ADVANCED
COMPUTING SYSTEMS
ASSOCIATION

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 technical sessions. This is perfect for folks on the go or those without consistent Internet access.

Box Sets are available for:

- **SREcon15**
- **FAST '15:** 13th USENIX Conference on File and Storage Technologies
- **LISA14:** 27th Large Installation System Administration Conference
- **OSDI '14:** 11th USENIX Symposium on Operating Systems Design and Implementation
- **TRIOS '14:** 2014 Conference on Timely Results in Operating Systems
- **USENIX Security '14:** 23rd USENIX Security Symposium
- **3GSE '14:** 2014 USENIX Summit on Gaming, Games, and Gamification in Security Education
- **FOCI '14:** 4th USENIX Workshop on Free and Open Communications on the Internet
- **HealthTech '14:** 2014 USENIX Summit on Health Information Technologies
- **WOOT '14:** 8th USENIX Workshop on Offensive Technologies
- **URES '14:** 2014 USENIX Release Engineering Summit
- **USENIX ATC '14:** 2014 USENIX Annual Technical Conference
- **UCMS '14:** 2014 USENIX Configuration Management Summit
- **HotStorage '14:** 6th USENIX Workshop on Hot Topics in Storage and File Systems
- **HotCloud '14:** 6th USENIX Workshop on Hot Topics in Cloud Computing
- **NSDI '14:** 11th USENIX Symposium on Networked Systems Design and Implementation
- **FAST '14:** 12th USENIX Conference on File and Storage Technologies
- **LISA '13:** 27th Large Installation System Administration Conference
- **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 Management 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

The Living Dead

TIM BRADSHAW

Tim Bradshaw was going to be a physicist before getting side-tracked by computers, in the form of UNIX and Lisp. He's worked on the border between programming and system administration for about 25 years. He worked in a large British retail bank during and after the crisis of 2007–2008 and is hoping not to be anywhere near one next time around. tfb@tfeb.org

It has been claimed [1] that system administration is dead: that may be so, but if you think that the dead are not a problem, you have been watching the wrong kind of movies. System administration may be dead, but it still walks.

The kind of system administration we'd like to be dead is *manual* system administration: the management of computing systems by people, rather than by programs. Not only is this very expensive but people are not up to the task: they make mistakes and forget things, so inconsistencies accumulate over time, leading to a slow collapse of the system. Long before the final collapse, it becomes impossible to deal with performance and security problems. These problems slowly kill organizations that try to run large systems using traditional approaches.

Why should we care? If we know how to do better now, then why not let organizations that cannot or will not learn fail? That's a solution only if we don't care about those organizations collapsing, and if we really do know how to do better.

I Walked with a Zombie

Consider an example of an organization that has solved the problem and one that hasn't: I'll pick Google for the former and a large retail bank of your choice for the latter. How bad would a week-long outage to each of these organizations be?

Google. This would be reasonably bad: search itself is a solved problem now—there are other adequate providers—but a lot of people have built their lives and businesses around products like Gmail without much thought. They'd have a bad week, and some businesses would die.

The Bank. This would be rather worse. If it was your bank, you would have no access to money at all other than cash you were carrying, and you would be hungry when that ran out. This may not be the end: banks are real-time organizations and can only be down for so long before they cannot recover, whether or not they have lost data. Opinions vary on how long this is but it's around a couple of days; your bank might never come back, and you would have to pick up the pieces of your financial affairs over months. This too may not be the end: the banking system is heavily interlinked, and the failure of a large retail bank could easily cause a cascade failure of other banks. The correct defense against that involves canned food, firearms, and growing a beard: you *do not* want a banking collapse to happen.

A failure like this is not easy to fix: you can bail out a bank that has run out of money by pouring money into it, but you can't bail out a bank whose computer systems have failed by pouring computers into it. We also should not assume that "someone else" will magically fix it for us. If the people who regulate banks were not competent [2] to see a rather obvious financial crisis coming in 2007–2008 until it was too late, they certainly are not competent to spot crises in computer systems, let alone fix them. And crises do happen. The 2012 RBS batch problem [3] was a damned near-run thing, and there is every reason to believe that something like it will happen again in a bank or some other equally critical organization.

Perhaps a banking collapse is not very likely, but it is definitely *possible*, and even a fairly small risk of the apocalypse is a thing to be avoided. Poor system administration practices matter, and it is not enough to declare them dead. We actually have to do something to stop them lurching around eating us.

Bone Sickness

We understand and can solve many of the roots of this malaise. Structural and funding problems and organizations outsourcing their own brains result from incompetent management. Administrators who do not program and programmers ("developers") who do not administrate lead to the problems you would expect, the solution to which is some variant of DevOps. Additionally, the people who build and deliver systems should be *the same people* who later maintain them; throwing rubbish over the wall is less appealing if you know you will have to clean it up later.

But these are not the only difficulties.

Old. The organizations I'm worrying about are *old*, which means they are not growing exponentially. All exponentially growing organizations are, effectively, young (although not all young organizations grow exponentially). Exponential growth famously kills companies, but there's a converse: if you *can* handle it, then all other problems are easy because all your mistakes get inflated away. Organizations growing exponentially can simply ignore old systems. Unfortunately, and despite what economists pretend to believe, exponential growth is necessarily ephemeral.

Contracts. If you have contracts with teeth, then you can't just turn off the system that is supporting a contract when it suits you. Dealing with this requires either applications and languages that are compatible for many years or physically supporting very old machines. The "Cascade of Attention-Deficit Teenagers" development model [4] means that the second option is often less bad, and we should be ashamed of this. Exponential growth inflates this problem away as well, while it goes on, but avoiding meaningful contracts is a clever trick if you can do it. Banks, sadly, are entirely made of contracts.

It is interesting that the canonical "good" organizations are exponentially growing, have avoided contracts with any real bite, and indeed do simply turn off services [5] when it suits them. Whether they really have solved the system administration problem will become more clear as their growth slows and contracts start to bite in the coming few years.

Power and Safety. To solve the system administration problem, you need powerful tools: tools that can influence very many machines, and tools that may themselves be computationally powerful and, hence, have behavior that is hard to reason about. Name services are an example of the former, and systems that can run arbitrary code on many machines, such as configuration management or patch deployment systems, are examples of the latter.

Such tools have inherent safety problems.

To start with, you need to be sure that whatever you are doing is either correct, does no harm if it is not correct, or, if harm is done, is fully reversible. Related to this are questions about authority and auditability: if you work for the sort of organizations we're discussing, you need to be able to show that you have authority to do something and later demonstrate convincingly to auditors that you had authority, that you actually did the thing you had authority to do, and so on.

Both of these problems exist already: a very powerful system simply makes them enormously more serious. It's the difference between the precautions you would take handling a stick of dynamite and handling an MK-53. These problems are mostly solvable in principle, although I don't think they are very close to being solved in practice. One non-solution is to divide the system up by some security mechanism so that large changes can't be made; well, yes, but then you will need lots of administrators for all the divided chunks, which is where we came in.

There is a graph that describes control and authority in a system: root nodes and nodes near them are extremely sensitive, as a compromise of them is a compromise of the system. Understanding the graph and working a lot harder on the security of the programs and protocols that sit at or near the roots of it would be a good start at dealing with this. Unfortunately, understanding the graph tells you one enormous thing: the roots are people and buildings, all of which can be attacked in very traditional ways, and the 2014 Sony attack [6] seems to be an example of that.

There are parts of the graph beyond any given organization that can themselves be compromised. For instance, the kernel.org compromise [7] was only not serious because it was discovered quickly and because of good engineering practices. Generally, there is blind faith that "vendor code," while probably buggy, won't be intentionally compromised or, if it is, only by the good guys [8]: why do we think that, since that code is right at the roots of the graph?

Banks are forced to care about these questions, and they are encrusted with auditors whose job is to make them care. They only have answers to some of them, and their answers tend to involve a deeply hideous bureaucracy. That very bureaucracy makes it extremely hard for them to think clearly about underlying problems such as the control and authority graph. This is particularly alarming given the sorts of configuration-management tools that they are being sold.

Warm Bodies

It is not just banks that are vulnerable, but utility companies, air traffic control, and governments: every organization whose failure would be most damaging. And bludgeoning the zombies isn't enough, since the problem is not really solved at all, other than in some rather special and almost certainly ephemeral cases. Do we really know how to manage large systems *in general* in a way that is demonstrably safe? Do we know how to build large systems that are safe at all? I don't think we do, not least because *really general solutions do not exist*. And claiming that we have solved problems that we, in fact, have not solved at all will simply suppress efforts to find answers that might be good enough. The obituary of system administration has been written prematurely.

References

[1] Todd Underwood, "The Death of System Administration," *;login:*, April 2014: https://www.usenix.org/publications/login/apr14/underwood.

[2] Sewell Chan, "Financial Crisis Was Avoidable, Inquiry Finds," *The New York Times*, January 25, 2011: http://www.nytimes.com/2011/01/26/business/economy/26inquiry.html.

[3] John Campbell, "Ulster Bank IT Problems: What Went Wrong," BBC News, November 20, 2014: http://www.bbc.co.uk/news/uk-northern-ireland-30127164.

[4] Jamie Zawinski, "The CADT Model," 2003: http://www.jwz.org/doc/cadt.html.

[5] "A Second Spring of Cleaning," Google, March 13, 2013: http://googleblog.blogspot.com/2013/03/a-second-spring-of-cleaning.html.

[6] Bruce Schneier, "Comments on the Sony Hack," December 11, 2014: https://www.schneier.com/blog/archives/2014/12/comments_on_the.html.

[7] Jonathan Corbet, "The Cracking of kernel.org," August 31, 2011: http://www.linuxfoundation.org/news-media/blogs/browse/2011/08/cracking-kernelorg.

[8] Bruce Schneier, "More About the NSA's Tailored Access Operations Unit," December 31, 2013: https://www.schneier.com/blog/archives/2013/12/more_about_the.html.

LISA15

More craft.
Less cruft.

Nov. 8 – 13, 2015 | Washington, D.C.

usenix.org/lisa15

The LISA conference is where IT operations professionals, site-reliability engineers, system administrators, architects, software engineers, and researchers come together, discuss, and gain real-world knowledge about designing, building, and maintaining the critical systems of our interconnected world.

Submit your ideas for talks, tutorials, panels, paper presentations, and workshops by April 17, 2015.

Topics of particular interest at this year's conference include systems and network engineering, monitoring and metrics, SRE/software engineering, and culture.

Sponsored by USENIX in cooperation with LOPSA

Software Testing for Sysadmin Programs

ADAM MOSKOWITZ

Adam Moskowitz is a Senior Tools Engineer at MathWorks and was the program chair of the LISA 2009 conference. Since entering the field in 1978, he has been a computer operator (what sysadmins were called in The Good Old Days), an application developer, a system administrator, and a teacher of all of those things. At MathWorks he is building a system to manage VMs in both production and ephemeral testing environments (and quietly take over the world). When he's not writing code, you will find him judging barbecue contests or riding a tandem bicycle with his wife. adamm@menlo.com

Testing is a common practice in modern software development, but the mere mention of it tends to raise hackles in the sysadmin community. To me, this is disappointing because testing produces better code and makes it easier to safely implement changes in your code. Too many articles about testing just argue its merits or excoriate people who don't use it, so instead I'm going to show you how easy it is to start testing your code and introduce you to a simple testing framework that is suitable for use with the kinds of programs sysadmins tend to write.

Before I go any further, let me explain exactly what I mean by testing. Good programs include defensive code—things like validating input, checking for errors, etc.; that is not testing. Rather, testing is writing a separate program that exercises your "real" code and proves it actually works. Typical test programs run your real program with a representative sample of valid and invalid inputs and verify that the responses from your program are correct. By having a separate program, you can make changes to your real code, and if all the tests still pass, it's likely that you didn't break anything.

System Administration Meets Software Development

Infrastructure as Code. You've all heard it before, and you probably have an idea of what it means. To me, one aspect of Infrastructure as Code is bringing the discipline of software engineering to bear on system administration. Software engineering covers a lot of ground, so let's focus on just three practices used by software engineers: version control, code reviews, and testing.

Pretty much everyone agrees version control is a good idea. You can use Git, Subversion, Perforce, or something else—just pick one and use it; all of them are better than not using any of them.

If you're doing code reviews, great; if not, they're trivial to start when you decide you're ready. I'm betting you can figure out how to do them without much help. You don't need a tool to do code reviews but using one can make the job easier. I like the tool ReviewBoard [4] but, again, there are plenty of other good tools.

Testing is where it gets interesting. Some of you may be testing your Puppet modules and Chef recipes, but what about all those other programs you wind up writing; do you write tests for those? To quote Yosemite Sam, "Dem's fight'n words!" Over the years, I've seen more and more sysadmins embrace the idea of using version control, and some are slowly embracing code reviews, but testing still seems to meet plenty of resistance.

As a software developer, my day looks something like this: Pick a task off the backlog (typically a task that needs to be automated), write some code and some tests (in either order or in parallel), keep working at it until the task is completed and all the tests pass, then submit a code review. Assuming my reviewer(s) liked what I wrote, I commit the code into our version control system. For the past year I've been writing most of my code in Groovy [2], using

Maven [3] to build my projects, and testing everything with Spock [5]; these were chosen as standards for my group before I was hired, and so far I haven't had any reason to change. All that was fine until one day I had to write a very "sysadmin-y" program in Bash, and the person reviewing my code said, "Where are your tests?"

Now, at my company, code reviews are pretty much required, but as the developer I have the option of choosing which changes requested by the reviewer I implement, including not implementing any at all. I would have been on solid ground to have said something like "No one writes unit tests for shell programs" or "It's too hard to write tests for shell," but that just didn't feel right to me. "Would it really be that hard to come up with some meaningful tests for this shell script?" I asked myself. I couldn't shake the feeling that I could come up with something that wouldn't feel like Rube Goldberg invented it, so I decided to spend a few hours on the problem.

By the end of the day I had enough of a skeleton developed that I believed I had found a viable solution; by the end of the next day I had a fully fleshed-out system, and the next morning (48 hours after the first code review), I submitted a second review with the comment, "Full tests for the Bash code are now included." Because I'm lucky enough to have time to take small detours when I find something that may be useful for my colleagues, I took a third day to build an example Bash program and set of tests to see just how far I could reduce the framework code. In the end I got it down to 57 lines and a very simple directory hierarchy. I'll show it to you in just a moment.

Arguments Against Software Testing

The most common reasons I've heard for not doing software testing fall into one of the following categories:

- Test harnesses are too hard to understand and too difficult to set up.
- It's too hard to test the kind of programs I write.
- It takes too much time.

Rather than get all preachy about it, I'm going to take the rest of this article to show you a technique I've been using for the past year that I believe will address at least the first two objections; I'll deal with the third objection later. Let's get to it, shall we?

Introducing a Testing Framework

There are more unit test frameworks than I can count. The Wikipedia article lists over 400 of them; Java alone has 35 different frameworks; Perl, Python, Ruby, and even Shell each have around eight. No wonder folks don't know where to start. As I wrote above, my solution is based on Groovy (which requires the Java JDK), Maven, and Spock; all are quite powerful and, in their full depth, are somewhat complicated, but I'm going to stick to a very small subset to keep things simple.

The setup is trivially easy: download the Java JDK gzipped tar file, unpack it somewhere (I like `/opt/<thing>-<version>` but you can do this all in `$HOME` if you prefer), make an optional symbolic link, and add one environment variable to your preferred shell's start-up file. Repeat for Groovy (a zip file) and Maven (requires two environment variables). When you're done, update your path. That's it: no installing dozens of packages, no dependency hell, no spending hours getting lots of little pieces all in the right places. The first time you test a program, Maven will automatically download Spock for you. A full set of detailed instructions can be found on my Web site [1].

The next step is to lay out your program source code, your test code, and tie it all together with a Maven `pom.xml` file. There's a link to these steps at [1], so I'll just show you what files go where:

```
-rw-r--r--  example/pom.xml
-rwxr-xr-x  example/src/main/bash/example
drwxr-xr-x  example/src/main/resources
-rw-r--r--  example/src/test/groovy/Test_example.groovy
drwxr-xr-x  example/src/test/resources
```

The file `example/src/main/bash/example` contains the program to be tested, and `example/src/test/groovy/Test_example.groovy` contains the test code; the former is written in Bash, the latter in Groovy. The two resource directories are there to keep Maven from complaining that they don't exist.

"Wait, Groovy?!?! I have to learn yet another programming language? Is this guy for real?" Please, there's no need to shout. No, you don't have to learn a new language, just a few constructs, and most of those are identical to Perl (which you probably already know); I'll give you enough examples of the new bits that I'm betting you'll be able to pick it up with very little work.

Here's our example program:

```
#!/bin/bash
if [ $1 = yes ] ; then
   echo hello, world
   exit 0
else
   echo goodbye, cruel world 1>&2
   exit 1
fi
```

As you can see it doesn't do anything useful, but it does just enough to let me demonstrate the three most basic Spock tests: testing the exit value of a program, examining standard out, and examining standard error. Here's the file `example/src/test/groovy/Test_example.groovy`:

```
1 package com.menlo.example
2
3 import spock.lang.*
4
5 class Test_example extends Specification {
6     static String here = System.getProperty("user.dir")
7     static String prog = "${here}/src/main/bash/example"
8
9     def "exits with 0"() {
10        when:
11            Process p = "${prog} yes".execute()
12            p.waitFor()
13
14        then:
15            p.exitValue() == 0
16            p.text.contains("hello, world")
17    }
18
19    def "exits with 1"() {
20        when:
21            Process p = "${prog} no".execute()
22            p.waitFor()
23
24        then:
25            p.exitValue() == 1
26            p.err.text.contains("goodbye, cruel world")
27    }
28 } // Test_example
```

For now, skip over lines 1–8; I'll cover them in the next paragraph. The first test is lines 9–17: line 10 defines the "stimulus" block, and line 14 defines the "response" block; that is, "given certain actions, confirm that certain results are true." In this case, run our program with the argument "yes" (line 11), then check that the exit value is zero (line 15) and that we get "hello, world" on standard out (line 16); you can ignore line 12 even though it's required. The second test (lines 19–27) is nearly identical to the first, the main difference being that we examine standard error instead of standard out (line 26).

For our purposes, that is, when testing programs written in anything other than Java or Groovy, lines 1–5 don't matter as long as they exist; it won't hurt anything if they're identical for every test file you ever write. Lines 6 and 7 define two variables that let us find our Bash program in a portable way; in other test files the only thing you'll need to change is `example` in line 7 to the name of the program being tested. (Obviously, if you're testing a Perl program, you'll also need to change `bash` to `perl`.)

To run the tests, just type `mvn test` and watch a few hundred lines of output go scrolling by; don't try to read it all, the important bits will be at the very end. You should see lines that look like this:

```
Running com.menlo.example.Test_example
. . .
Tests run: 2, Failures: 0, Errors: 0, Skipped: 0
. . .
[INFO] BUILD SUCCESS
```

What I haven't shown you, and what I'm going to ask you to take on faith, is how Maven knows what to do. It's all contained in the file `pom.xml`, and the example you can download from my Web site has everything you need; there's no need to modify the file or even look inside it; just put a copy in the top directory of each Maven project and you'll be fine. My Web site also contains a small shell script, `new-testing-project`, that will create new Maven project directories for you, populate them with skeleton files, and drop a fully formed `pom.xml` into place.

Testing Scripts

For this next bit, let's agree to call the programs that sysadmins write "scripts," only because it will provide a convenient shorthand; also, let's agree to call things like `mount` and `ifconfig` "system programs," again, for convenience.

One of the challenges with testing scripts is they often rely heavily on system programs; a typical script takes an argument or two from the command line, uses it to call a system program, captures the output of that program, manipulates it, then passes it to a second system program. Most of the tricky code in the script is dedicated to parsing the output of the system programs, trying to extract the desired pieces or detect an error. In many cases, actually running the system programs is destructive or produces an undesirable result, or there's no easy way to cause the system program to fail (so the script's error-checking code can be, um, checked). Fortunately, there's a technique common in modern software development that can be used, after a fashion, for testing scripts as well; this technique is often known as "mocks" or "stubs."

USING MOCKS IN PLACE OF SYSTEM COMMANDS

The idea behind a mock is simple: rather than run `/bin/mount`, we run our own private imitation of `mount` that emits whatever output we need for our tests but doesn't affect the state of the system. Some scripts call system programs via an explicit path—for example, `/bin/mount -l -t ext3`—instead of relying on `mount` being somewhere in `$PATH`. There's something to be said for that style but it makes testing impossible. While you could depend on `$PATH`, making sure to set it at the top of every script you write, the alternative I prefer is to use variables for each system program I call. By doing it like this:

```
MOUNT=${TESTBIN:-/bin}/mount
```

you're still protected from an incorrectly set `$PATH` while, at the same time, having the flexibility to run a mock during testing.

Here's a second example that uses mocks:

```
-rw-r--r--  example2/pom.xml
-rwxr-xr-x  example2/src/main/bash/example2
drwxr-xr-x  example2/src/main/resources
-rw-r--r--  example2/src/test/groovy/Test_example2.groovy
drwxr-xr-x  example2/src/test/resources
drwxr-xr-x  example2/src/test/resources/bin
-rwxr-xr-x  example2/src/test/resources/bin/mount
drwxr-xr-x  example2/src/test/resources/data
-rw-r--r--  example2/src/test/resources/data/mount.error
-rw-r--r--  example2/src/test/resources/data/mount.single
-rw-r--r--  example2/src/test/resources/data/mount.separate
```

The files `mount.single` and `mount.separate` contain the output from `mount` on systems with everything on a single partition and with /, `/home`, `/tmp`, `/usr`, and `/var` on separate partitions.

Our mock `mount` command is trivially short:

```
#!/bin/bash
cat $MOCK_MOUNT_FILE
exit $MOCK_MOUNT_EXIT
```

Obviously, if your script calls `mount` more than once in a single run, a more sophisticated mock is needed. Finally, here's how we tie all this together inside `Test_example2.groovy`:

```
package com.menlo.example2

import spock.lang.*

class Test_example2 extends Specification {
   static String here    = System.getProperty("user.dir")
   static String bin     = "${here}/src/test/resources/bin"
   static String data    = "${here}/src/test/resources/data"
   static String prog    = "${here}/src/main/bash/example2"
   static String wrapper = "${here}/target/example2"

   def "test all on one partition"() {
   setup:
      File f = new File(wrapper)
      f.delete() // make sure we start with a new file
      f << "#!/bin/bash\n"
      f << "export MOCK_MOUNT_FILE=${data}/mount.single\n"
      f << "export MOCK_MOUNT_EXIT=0\n"
      f << "export TESTBIN=${bin}\n"
      f << "${prog}\n"
      f.setExecutable(true, false)

   when:
      Process p = "${wrapper}".execute()
      p.waitFor()

   then:
      // your tests here

   cleanup:
      assert new File(wrapper).delete()

   }
} // Test_example2
```

Each subsequent test would have to duplicate the `setup` and `cleanup` stanzas, substituting values for `MOCK_MOUNT_FILE` and `MOCK_MOUNT_EXIT` as appropriate.

You may have noticed the directory `target` in the definition of `wrapper`; this is where Maven puts all temporary files. When you're done testing a particular script, run the command `mvn clean` to clean up.

Mocks can be quite complicated, and I could probably fill an entire article on how to get fancy with them; for now I think I've left you with enough to get started.

"It Takes Too Long"

The last reason people give for not writing tests—"it takes too much time"—is often the most difficult to respond to, but I hope by now I've established enough credibility that you'll at least read my argument.

To me, testing is kind of like insurance: if you never need it then the money you spend on it is "wasted," but it takes only one accident (or failure or whatever) for every dollar you've paid in premiums to be returned ten-fold (or more). But software will inevitably fail, and writing software is hard. There are far more variables, edge cases, and unknowns involved in software than any one person can understand, and even the best developers are far from perfect. Put together, it's not a question of whether any given piece of software will fail but, rather, when it will fail and how much damage the failure will cause.

Of course, tests are themselves software and thus will fail, but my 30+ years of experience tells me that the time typically put into writing tests catches at least 80% of the bugs; I also know that writing tests makes you approach software development differently and results in more correct (or, at least, more robust) software. To me, producing better software far outweighs the fact that testing is not a "silver bullet" and that your software may still fail. In other words, apply the Pareto Principle (aka "the 80-20 rule") and avoid letting perfection get in the way of improving your work.

The other big benefit to testing is that it lets you make changes to your software; not only can you tell that the new code works, you can be confident that you haven't broken your old code. For example, if you wrote a program to run under CentOS but now want it to work on Ubuntu as well, you can run the tests to prove it works on CentOS, modify the code, and add tests for Ubuntu. Once the new stuff is working, go back and run the old tests on CentOS to see that your program still works there.

So there you have it: I've shown you how to install and configure a simple test harness, and how to write basic tests; I've given you a brief introduction to mocks; and I've offered an argument to justify spending the extra time to write tests for your programs. Now it's up to you to decide whether to apply what I've shown you the next time you have to write a program as part of your job as a system administrator.

Resources

[1] http://menlo.com/tdd4sa/.

[2] http://groovy.codehaus.org/.

[3] https://maven.apache.org/.

[4] https://www.reviewboard.org/.

[5] https://code.google.com/p/spock/.

XKCD

Managing Incidents

ANDREW STRIBBLEHILL AND KAVITA GULIANI

Andrew Stribblehill has been a Site Reliability Engineer at Google since 2006 and has worked on ad serving, database, billing pipeline, and fraud detection teams. Before joining Google, he studied theoretical physics at Durham University after which he specialized in high-performance computing for the university. stribb@google.com

Kavita Guliani is a Technical Writer for Technical Infrastructure and Site Reliability Engineering at Google Mountain View. Before working at Google, Kavita worked for companies like Symantec, Cisco, and Lam Research Corporation. She holds a degree in English from Delhi University and studied technical writing at San Jose State University. kguliani@google.com

Effective incident management is key to limiting the disruption caused by an incident and restoring normal business operations as quickly as possible. Without gaming it out in advance, and with adrenalin running through the veins, principled incident management can go out of the window in real-life situations.

This article walks you through a pen portrait of an incident that spirals out of control due to ad hoc incident management practices, outlines a well-managed approach, and reviews how the same incident might have looked with well-functioning incident management.

Unmanaged Incidents

Put yourself in the shoes of Mary, the on-call engineer for The Firm. It's 2 p.m. on a Friday and your pager has just exploded. Black-box monitoring tells you that your service has stopped serving any traffic in an entire datacenter. With a sigh, you put down your coffee and set about the job of fixing it. A few minutes into the task, another alert tells you that a second datacenter has stopped serving. And then a third, out of five. Worse, there is more traffic than the remaining two datacenters can handle; they start to overload. Before you know it, your service is overloaded and unable to serve any requests.

You stare at the logs for what seems like an eternity. There are thousands of lines that suggest there's an error in one of the recently updated modules, so you decide to revert the servers to the previous release. When you see that this hasn't helped, you call Josephine, who wrote most of the code for the now-hemorrhaging service. Reminding you that it's 3:30 a.m. in her time zone, she blearily agrees to log in and take a look. Your colleagues Sabrina and Robin start poking around from their own terminals. "Just looking," they tell you.

Now one of the suits has phoned your boss and is angrily demanding to know why he wasn't informed about this "business-critical service's total meltdown." Independently, the vice presidents are nagging you for an ETA and to understand "How could this possibly have happened?" You would sympathize but that takes cognitive effort that you are holding in reserve for your job. They call on their prior engineering experience and make irrelevant but hard-to-refute comments: "Increase the page size" sticks in the mind.

Time passes and the two remaining datacenters fail completely. Unbeknown to you, the sleep-addled Josephine had called Malcolm. He had a brainwave: something about CPU affinity. He felt certain that he could optimize your remaining server processes if he could just deploy this one simple change to the production environment, so he tried it. Within seconds, the servers restarted, picking up the change. And died.

The Anatomy of an Unmanaged Incident

Note that everybody in the above picture was doing their job, as they saw it. How could things go so wrong? Here are some hazards to watch out for:

Sharp Focus on the Technical Problem

We tend to hire people like Mary for their technical prowess. So it's unsurprising that she was busy making operational changes to the system, trying valiantly to solve the problem. She wasn't in a position to think about how to mitigate the problem: the technical task at hand was overwhelming.

Poor Communication

For the same reason, she was far too busy to communicate clearly. Nobody knew what others were doing. Business leaders were angered; customers frustrated; the would-be volunteers switched off or, at any rate, were not used effectively.

Freelancing

Malcolm was making changes to the system with the best of intentions. But Malcolm hadn't coordinated his actions with anyone, perhaps because no one, not even Mary, was actually in charge of troubleshooting. Whatever the case, his changes made a bad situation far worse.

Elements of Incident-Management Process

Incident-management skills and practices exist to channel the energies of enthusiastic individuals. Google's incident-management system is based on the Incident Command System [1], known for its clarity and scalability.

A well-designed incident-management process has the following features:

Recursive Separation of Responsibilities

It's important to make sure that everybody involved in the incident knows their role and doesn't stray onto someone else's turf. Somewhat counterintuitively, this allows people more autonomy than they might otherwise have since they need not second-guess their colleagues.

If the load on a given member becomes excessive, they need to ask the planning lead for more staff, then delegate work, up to and including the creation of sub-incidents. Less extreme delegation involves role leaders factoring out noisy components to colleagues who report high-level information back up to them.

There are several distinct roles that are both easy to identify and sufficiently separable that they can be worked on by different people:

Incident Command: The incident commander holds the high-level state about the incident. They structure the incident response task force, assigning responsibilities according to the need. De facto, they hold all positions that they have not delegated. If appropriate, they can suggest ways around problems to Ops, thus helping them to avoid fixating on unhelpful parts of the problem space.

Operational Work: The Ops lead works with the incident commander to respond to the incident by applying operational tools to the task at hand.

Communication: People filling this role are to be the public face of the incident-response task force. They might take on the synopsis-writing component of maintaining the incident document; they almost certainly send emails periodically to keep others aware of the progress made.

Planning: Dealing with longer-term issues than Ops, people in the Planning role support Ops, match offers of support with roles, file bugs, and track how the system has diverged from the norm so that it can be reverted once the trouble is over.

A Recognized Command Post

Interested parties need to understand where they can interact with the incident commander. In many situations, taking the incident task force members into a known room is effective; scrawl "War Room" on the door if that helps. Others may prefer to remain at their desk but keep a weather eye on email and IRC.

We have found IRC to be a huge boon in incident response. It is very reliable and can be used as a log to scroll back through, which is invaluable in keeping detailed state changes in mind. We've written bots that log the traffic (helpful for postmortem analysis) and others that report interesting events such as alerts to the channel. IRC is also a convenient medium over which geographically distributed teams can coordinate.

Live Incident State Document

The most important responsibility of the incident commander is to keep a living incident document. This can be in a wiki but it helps if it's editable by several people concurrently. Most teams at Google use Google Docs for this, although Google Docs SREs use Google Sites: after all, depending on the software you are trying to fix as part of your incident-management system is unlikely to end well.

See Appendix 1 for a sample incident document. They can be messy but they must be functional, so be sure to make a template first so that it's easy to start your incident documents. Keep the most important information at the top. Retain them for postmortem analysis and, if necessary, meta-analysis.

Clear, Live Handoff

It's essential that the post of incident commander be clearly handed off at the end of the working day. If outgoing and new people are not both in the same location, a simple and safe way is to update the new incident commander over the phone or video call. Once the new incident commander is fully apprised, the outgoing commander should explicitly make the handoff ("You're now the incident commander; okay?") and not leave the call until there is firm acknowledgment.

A Managed Incident

Mary is into her third coffee of the day: it's 2 p.m. The pager's harsh tone surprises her and she gulps the drink down. Problem. A datacenter has stopped serving traffic. She starts to investigate. Shortly another alert fires and the second datacenter out of five is out of order. Since this is a rapidly growing issue, she knows that she'll benefit from the structure of her incident-management framework.

Mary snags Sabrina. "Can you take command?" Nodding her agreement, Sabrina quickly gets a rundown of the story so far, writes it up as an email, and sends it to the prearranged mailing list. Sabrina recognizes that she doesn't know the impact yet, so she asks Mary. "None so far. Let's just hope we don't lose a third datacenter." She copies it into a live incident document, marking it as the summary.

When the third alert fires, Sabrina sees it among the debugging chatter on IRC and quickly follows up to the email thread with an update: the VPs are being informed about the high-level status of the incident without being bothered with the minutiae. She seeks out an external communications representative so they can start drafting their messaging. "How's it going, Mary?" asks Sabrina. "Want me to find the developer on call?" "Sure."

By the time Josephine logs in, Robin has already volunteered to help out. Sabrina reminds him and Josephine that they are to keep Mary informed of any actions they are taking unless Mary delegates to them. They quickly learn the current situation by reading the incident document.

Mary has, by now, tried the old binary release and found it wanting: she mutters this and Robin updates IRC to say that it didn't work. Sabrina pastes it into the document.

At 5 p.m., Sabrina starts finding replacement staff to take on the incident: she and her colleagues are about to go home. Sabrina updates the incident document. At 5:45 she holds a brief phone conference to make sure that everyone's aware of the current situation. By 6 p.m., they're all tired, and they hand off their responsibilities to their colleagues in the remote office.

When to Declare an Incident

It is better to declare an incident early and then find a simple fix and shut it down than to have to spin up the incident-management framework hours into a burgeoning problem. Set clear conditions for declaring an incident. My team follows these broad guidelines. If any of the following is true, it's an incident:

- Do you need to involve a second team in fixing the problem?
- Is it a customer-visible outage?
- Is it an unsolved issue even after an hour's concentrated analysis?

Incident-management proficiency atrophies quickly. So what is an engineer supposed to do? Have more incidents? Fortunately, the incident-management framework can be followed for other operational changes that need to span time zones and/or teams. If you use it frequently as a regular part of your change-management procedures, it'll be easy to pick up when needed. If your organization performs disaster-recovery testing (you should: it's fun), incident-management should be part of that process. We often game out the response to an on-call issue to further familiarize ourselves.

Resource

[1] https://www.osha.gov/SLTC/etools/ics/what_is_ics.html.

Appendix 1: Incident Template

Incident <WRITE TITLE HERE>: yyyy-mm-dd

Incident management info: http://<your-internal-site>/incident-management-cheat-sheet

~TWO-LINE SUMMARY; COMMS-LEAD TO MAINTAIN CONTINUALLY.

Status: active
Command post(s): #incident on IRC

Command Hierarchy (all responders)

- Current Incident Commander: XXX
  - Operations lead: XXX
  - Planning lead: *role filled by Command*
  - Communications lead: *role filled by Command*
- Next Incident Commander: *undefined*

Detailed Status (updated at yyyy-mm-dd hh:mm UTC by $user)

UPDATE EVERY 4 HOURS AND AT HANDOFF

Exit Criteria

TODO list and bugs filed

Incident timeline, most recent first: times are in UTC

yyyy-mm-dd
hh:mm USERNAME: XXX

/var/log/manager
A Generational Theory of Sysadmins

ANDY SEELY

Andy Seely is the Chief Engineer and Division Manager for an IT enterprise services contract, and an adjunct instructor in the Information and Technology Department at the University of Tampa. His wife Heather is his PXE Boot and his sons Marek and Ivo are always challenging his thin-provisioning strategy. andy@yankeetown.com

System administrators start from someplace. They're born to it, they're trained for it, or maybe they just fall into it. Regardless of the origin story, each sysadmin brings a certain something, a fire in the belly, an approach that is creative and inquisitive, and a motivation that is oddly tough to pin down. Over the past few years of hiring technical staff, I started getting a feeling that the type of sysadmin I was used to finding just wasn't as common any more. I felt like something was changing in the work force, which led me to develop my own "theory" of generations of sysadmins and ultimately to change how I approach hiring and managing sysadmins who don't fit into just one model.

The Explorer

Before my own career logs were being written into /var/log/manager, I was a pure and serious sysadmin. I wasn't just the master of esoterica, I was frequently the only person in the whole operation who knew certain things. I was the only one who knew what a "magic number" was. I was the only sysadmin who could write code in FORTH, and I applied that skill to write custom boot orders for Sun servers. I saw operating systems, databases, HTTP, SMTP, NNTP, IRC, TCL/TK/Expect, Perl, and Motif as all dialects of the same underlying language, and I spent incredible amounts of time burrowing into the oddest little artifacts of an operating system and gaining the most intimate understanding of every system and how it interacted.

My own electricity bill was a monthly casualty of these explorations. I didn't do all this exploring and tinkering on company time, although I didn't not do it on company time either; I spent considerable home time and personal expense building out servers and labs and filling rooms with boxes and cables. It's remarkable that my wife is still with me after I spent the first decade of our marriage duct-taping Ethernet cables across the floor. I was incredibly personally invested.

I was the young guy on the team. The older folks, my mentors and managers, didn't seem to have had my type of personal, home investment when they were younger. Something about needing whole power substations to run a home lab made out of Honeywell gear, or the expense of buying your own PDP-11. Apparently, a nine-track tape cabinet is a serious spouse irritant. That didn't stop them from being brilliant contributors on the job and fine mentors to young kids like me, but they had obviously learned the trade differently than I was doing.

The Wall of Certs

Today, I'm the old guy, and I interview young people looking for jobs. I'll find people who have home labs, but it's usually to build out an environment in order to study for a certification exam. This means that a home lab is constructed to install a trial version Microsoft Windows Server 2008R2 with SQL Server and Active Directory. It's a good drill and has a positive result when it comes to test time, but you don't find as many young people today running a half-dozen different operating systems at a time just to see which one is the most "fun." To

a non-sysadmin, that kind of activity probably seems wasteful and more than a little useless, but to me the benefits of understanding how an OS performs can't be replicated easily in other ways. "Fun" is a feature of the learning.

Generations of Sysadmins

This slow change in the type of applicants I see has had me thinking about what's changed. Perhaps there are distinct generations of sysadmins. Making a notional model of something poorly understood helps with the understanding; we do this all the time at the whiteboard when troubleshooting. A little bit of time at my manager's whiteboard produced an informal "Generational Theory of Sysadmins."

1. The Baby Boomers learned about computing in college, were trained as electrical or mechanical engineers, or happened to be serious workshop hobbyists. Being a sysadmin meant having a soldering iron and an oscilloscope. They bought kits or built rudimentary computers in garages and helped spark the personal computing market. These are the senior sysadmins and managers who are starting to retire. I think of them as "Builders."
2. Generation Xers grew up with those Boomers. This was the first generation to be exposed as kids to personal computing. There were commercial boxes you could buy and games to play, but it was still in the "some assembly required" space. A kid might have to type in a thousand lines of BASIC from a magazine in order to play a game, but ultimately there was a computer and a computer game in the home, and that pushed the kid into typing it in and, eventually, figuring it out. These are today's mid-career sysadmins and managers who are hiring and mentoring the next generation. I started out thinking of them as "Hackers," but that's a word that means too many different things to people for it to be the best word. "Explorers" is a better word.
3. Millennials, aka Generation Y, are in the early stages of their careers. They're college graduates and certification-holders, and they grew up in households where computers were bought but not built, and where software was bought or downloaded and installed, but not necessarily written and certainly never typed in from a magazine. Even the more technical Gen-Y people still see computing as something to assemble and use rather than something to build and figure out. I think of them as "Users of Tools."
4. Generation Z is expert with modern technology. I can install a new game app on my smartphone and hand it to my four year old, and with zero training he can be up and running in barely minutes ("Dad, this is awesome!" said the boy two minutes after I handed him "Cut the Rope"). Yet it's misguided to think of these youngsters as technologically savvy. What they have is the ability to intuitively grasp today's well-designed UIs and easily approach communications as an abstract concept. They've grown up hearing grandma's voice coming out of the ether of a vehicle's Bluetooth connection to the mobile phone, and they're perfectly comfortable with things happening to and for them, but they don't necessarily gain any understanding of how these things happen. I think of them as "Customers of the Future."

There's a danger in creating any labeling system for people. It dehumanizes. It pigeonholes and silos. It creates friction points and us-versus-them scenarios. And it allows a manager to take an easy view of things and assume understanding of a person simply based on a label, which is dangerous ground. In my sysadmin generational theory, there's a lot of overlap and slop around the edges. It's simply a way of thinking about people's skills in order to help guide them, and nothing more.

The Future of Sysadmin

As a manager, I look for traits in prospective employees that I see in myself: personal investment in the sysadmin profession, drive, motivation, curiosity, and a burning need to understand how something works behind the UI. I'm seeing less and less of that in younger applicants, and I'm concerned. I'm also growing as a manager and learning how to see different kinds of value in people.

As the manager, it's my job to understand the capabilities of my employees and play up the strengths while avoiding the weaknesses. My generational model helps me find the value in people who don't have the same mindset I do. I'm the manager, and organizing people of different skills and generations into effective and efficient teams is my job.

A Tribute

Where do I fit and how did I get here? I'm a pure Gen-X "Explorer" sysadmin. I grew up with a Builder-type dad, who brought home what may have been the first personally owned computer in my hometown in 1980. He was never a computing professional, but he always kept the latest in our house and always made sure that we were growing up with technology at our fingertips. The timing was perfect, as the technology was raw enough that I needed to tinker to make anything work. Thanks to a good 10 years of my father's Boomer-style hobbyist obsession with computing, I was able to make an easy jump into an entry-level computing job, and then I took off and never slowed down.

This sidebar is a tribute to my father, Ray Seely, who died too young on December 17, 2014.

Practical Perl Tools

Dance, Browser, Dance!

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@pobox.com

In past columns, we've written code together that contacted Web sites that didn't have an API per se and queried information from them. Tools like HTTP::Tiny, LWP::Simple, Mojo::UserAgent, and WWW::Mechanize have made an appearance in this column (some as recently as the previous column). These are all fantastic tools (some of them more fantastic than others), but if you have felt something was lacking, I can't blame you. With all of these modules, we've sidestepped, for better or worse, the Web browser. This has also meant giving up certain functionality found in the browser—the biggest elephant being JavaScript. People have written code to glue JavaScript engines to WWW::Mechanize (e.g., WWW::Mechanize::PhantomJS) or to drive browsers from these kinds of modules, but they haven't been particularly widespread in their implementation or adoption. In this column, we're going to look at how to use Perl with a framework that lots of people use to drive browsers in a whole range of languages.

The framework we'll be exploring in brief today is called Selenium. It originated from work that people have done to create testing frameworks that used real browsers to construct real tests of Web applications. Let's say you built a Web app and you'd like to make sure that your automated test suite (you have a test suite, right?) actually tests the app's functionality using the same browsers humans will be using when you finally make it available on the Web. Enter Selenium (http://www.seleniumhq.org). But, testing is just one thing you could use this for; driving your browsers (both desktop and mobile) using a script could be applied to all sorts of things.

Before we dive into how to set all this up and get it to rock from Perl, there is a piece of Selenium history worth mentioning so that you don't take a wrong turn while learning about this stuff. Once upon a time, as in version 1, Selenium offered something called Selenium Remote Control (or Selenium RC as you will see it written) as one of its main interfaces.

There were a number of Perl modules written for Selenium 1, and we're not going to touch any of them. Selenium 1 was a bit of a hack (basically it injected JavaScript code that manipulated the browser), so at some point Selenium 2 (sometimes called Selenium WebDriver because that was the name of the other project that merged with Selenium) was born. In this column, we are going to be using a Perl module that works with Selenium 2 only. If you want to dive deeper into this subject by searching the Web for more information, be sure to pay attention to which version of Selenium the resources you find are describing.

Wait, Was that Java I Just Saw Zoom By?

Let's talk about how we get set up to start using Selenium. While there are ways to directly talk to a browser using the WebDriver stuff, the Perl module we're going to be using expects to talk to a standalone Selenium server. That server is written in Java. But, besides needing the JDK installed and running one command, you can pretend I never mentioned that language. Actually, let me be a little bit of a tease and mention that there are companies like Sauce Labs (https://saucelabs.com) that actually provide Selenium as a service so that you could connect

to their hosted Selenium infrastructure instead of bringing up your own server. But for our purposes, bringing up a standalone server (vs. an industrial-strength service) is pretty trivial.

First step (providing you have the JDK installed): go to http://www.seleniumhq.org and download the latest stable Selenium server release.

Step 2: start it up like so:

```
java -jar {name of jar file}
```

See, that wasn't so bad. The Java app will produce output that will look roughly like this:

```
21:52:09.373 INFO - Launching a standalone server
21:52:09.437 INFO - Java: Apple Inc. 20.65-b04-466.1
21:52:09.437 INFO - OS: Mac OS X 10.10.1 x86\_64
21:52:09.458 INFO - v2.44.0, with Core v2.44.0. Built from
revision 76d78cf
21:52:09.574 INFO - Default driver org.openqa.selenium.
ie.InternetExplorerDriver registration is skipped:
registration capabilities Capabilities \[{platform=WINDOWS,
ensureCleanSession=true, browserName=internet explorer,
version=}] does not match with current platform: MAC
21:52:09.643 INFO - RemoteWebDriver instances should connect
to: http://127.0.0.1:4444/wd/hub
21:52:09.644 INFO - Version Jetty/5.1.x
21:52:09.645 INFO - Started HttpContext\[/selenium-server
/driver,/selenium-server/driver]
21:52:09.646 INFO - Started HttpContext\[/selenium-server
,/selenium-server]
21:52:09.646 INFO - Started HttpContext\[/,/]
21:52:09.717 INFO - Started org.openqa.jetty.jetty.servlet
.ServletHandler@3b6f0be8
21:52:09.717 INFO - Started HttpContext\[/wd,/wd]
21:52:09.727 INFO - Started SocketListener on 0.0.0.0:4444
21:52:09.728 INFO - Started 657576922 org.openqa.jetty.jetty
.Server@7a3570b0
```

This output will be primarily useful to us if we want to check some of the values in use (e.g., what port it is listening on). A number of values can be set on start; to see what is supported, run the following:

```
java -jar {name of jar file} -help
```

Back to Cool, Refreshing Perl

Once you have a Selenium standalone server running, it is time to bring Perl into the picture. The module we are going use is called Selenium::Remote::Driver. It can be a little dependency heavy (48 other modules if installing into a fresh Perl instance—I checked), but with the help of the cpanm command mentioned here in a past column, it is installed with a single command (cpanm Selenium::Remote::Driver) and a bit of thumb-twiddling.

Let's start with a simple script that uses it to tell a browser to fetch a Web page:

```
use Selenium::Remote::Driver;

my $driver = new Selenium::Remote::Driver;
$driver->get('http://www.usenix.org');
print $driver->get_title(),"\n";
$driver->quit();
```

This script can be so bare bones because it is using all of the defaults; we'll talk about modifying them shortly. When we run this script, it is a little creepy because Firefox pops open, loads this page, quits, and then the script prints:

```
Home | USENIX
```

We can see what is going on because the window where we started the standalone server is providing some play-by-play debug output:

```
11:36:19.871 INFO - Executing: [new session:
Capabilities [{acceptSslCerts=true,
browserName=firefox, javascriptEnabled=true, version=,
platform=ANY}]])
11:36:19.894 INFO - Creating a new session for Capabilities
[{acceptSslCerts=true, browserName=firefox,
javascriptEnabled=true, version=, platform=ANY}]
11:36:26.008 INFO - Done: [new session: Capabilities
[{acceptSslCerts=true, browserName=firefox,
javascriptEnabled=true, version=, platform=ANY}]]
11:36:26.018 INFO - Executing: [get: http://www.usenix.org])
11:36:28.186 INFO - Done: [get: http://www.usenix.org]
11:36:28.192 INFO - Executing: [get title])
11:36:28.536 INFO - Done: [get title]
11:36:28.542 INFO - Executing: [delete session: 1abb3d91
-ce4a-426d-8096-b4853cf94197])
11:36:28.646 INFO - Done: [delete session: 1abb3d91-ce4a
-426d-8096-b4853cf94197]
```

You Can Seek, But First You Have to Find First

Retrieving the title of the page you opened in the browser via Selenium magic is probably not the most useful thing you will want to do (although it can be helpful as part of a larger test suite to make sure the rest of the code's assumptions about which page you're on are correct). Most of the time, you will want to be working with elements on that page, either retrieving them or interacting with them (e.g., filling in forms, performing some kind of navigation).

More often than not, the very first thing you have to do is grab hold of part of the page using one of these two find_ commands:

find_element
find_elements

There are other similar commands (e.g., get_active_element, which returns the element that has focus), but I find almost all of my scripts include one of those two as the first action after pulling up the page.

Here's where things get a little interesting and where one of the defaults mentioned before comes into play. find_element(s) gives you three different "strategies" (that's the term from the docs) for locating elements:

1. HTML specification (my term). This lets you find an element by id (*id = something* in the source)), class (*class = something*), link, etc.
2. CSS specification. This lets you find an element using the standard CSS selectors as the browser implements them. So, for example, you could specify "div#feature3".
3. XPath specification. Faithful readers of this column know I ♥ XPath for its concision and eloquence. They will also recall that we spent an entire column looking at the XPath syntax and like. So that we don't have to do an entire context swap-in of that info, I'm going to simply say that one can use XPath expressions as another way of selecting elements on a page but not provide other examples of this that need to be explained.

By default, find_element will use #3, XPath. To change that default, each find_element can take a second argument specifying the strategy, or better yet, we can change the default:

```
my $driver =
  Selenium::Remote::Driver->new('default_finder' => 'css');
```

The docs recommend using HTML selectors (#1) by default, as in:

```
my $Webelement = find_element('search-bar','id');
```

because it is the most efficient kind of search, but that assumes you are dealing with Web pages that have well-structured code. I tend to hope for that but expect to have to use one of the other kinds of finders.

Now we know ways to find things, but what happens when we succeed? find_element() will return a WebElement object (or more precisely, a Selenium::Remote::WebElement object) representing the first thing it finds, and find_elements returns an array of them for all of the matches. With this object, we can do a number of things (documented in the Selenium::Remote::WebElement module documentation). Here's some code that will display the names of the main tabs on the page:

```
my (@elements) =
 $driver->find_elements('ul#main-menu-links li a','css');
foreach my $element (@elements){
 print $element->get_text(),"\n";
}
```

It queries for all of the elements that match a particular CSS selector (finds all of the links in the list items of the unordered list with the ID of "main-menu-links") and then displays the text associated with each.

Let's Do Stuff

Selenium has launched a browser for us, so let's start doing browse-y things. First off, we might want to start navigating around the page and clicking on stuff. One thing we could do would be to click on all of the main menu tabs and retrieve the page title for each page we land on. Let's start with code that does not work, because it will illustrate an important point:

```
# this does not work!
my (@elements) =
  $driver->findelements('ul#main-menu-links li a','css');

# the first link is to the current page, skip it
shift @elements;
foreach my $element (@elements){
 $element->click();
 print $driver->gettitle(),"\n";
 $driver->goback();
}
```

This would seem to be the right thing. Find all of the links, click on a link, hit the back button, click on the next link, easy, right? Here's what happens when we run the code:

```
About USENIX | USENIX

Error while executing command: An element command failed
because the referenced element is no longer attached to the
DOM.: Element not found in the cache - perhaps the page has
changed since it was looked up
...
```

It gets the first click/title print right, but bites the dust on the second one. Why is that? In this case, we've clicked to another page before coming back to the home page. When we return to the home page, there's no guarantee that the structure of the page (the DOM to be precise) we return to is exactly the same as the way we left it. Lots of stuff could happen—the source of the page could have been changed, JavaScript could have altered the structure, and so on. Selenium knows we are dealing with essentially a new page, so the references to parts of the old page aren't viable anymore. The best we can do is rerun the find_elements() and pick the next item in a list whose index we retain. Here's code that does work:

```
my (@elements) =
 $driver->find_elements('ul#main-menu-links li a','css');
```

```
for (my $tab = 1; $tab <= $#elements;$tab++){
 $elements[$tab]->click();
 print $driver->get_title(),"\n";
 $driver->goback();
 @elements =
$driver->find_elements('ul#main-menu-links li a','css');
}
```

If we run it, we get the following output:

```
About USENIX | USENIX
Conferences | USENIX
Publications | USENIX
LISA Special Interest Group for Sysadmins | USENIX
Membership & Services | USENIX
Student Programs | USENIX
USENIX | The Advanced Computing Systems Association
```

Basically, we do another find each time we return to the home page and then click on the next tab in the sequence. Long-time programmers are probably reaching for their pitchforks because they can smell a race condition when they see one, so let me cop to it right now. Yup, this code could potentially lead to a race condition. As in the vaudeville skit where the patient says, "Doctor, Doctor, please help me, it hurts when I move my arm like this," the response is "Don't move your arm like that."

Go Forth and Do Cool Stuff

As you can probably guess, Selenium has lots of other actions you can take on a page. You can select elements, you can send key presses, drag and drop, move the mouse around, select different windows, and so on. In addition to the documentation, there are two good tutorials at http://www.slideshare.net/vroom/testing-your-Website-with-selenium-perl and http://desmoines.pm.org/meetings/selenium_july2013.html worth checking out. Enjoy, and we'll see you next time.

Raising Hell, Catching Errors

DAVID BEAZLEY

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

One of my favorite Python topics to talk about is error handling—specifically, how to use and not use exceptions. Error handling is hard and tricky. Error handling can mean the difference between an application that can be debugged and one that can't. Error handling can blow your business up in the middle of the night if you aren't careful. So, yes, how you handle errors is important. In this article, I'll dig into some of the details of exceptions, some surefire techniques for shooting yourself in the foot, and some ways to avoid it.

Exception Handling Basics

To signal errors, Python almost always uses exceptions. For example:

```
>>> int('42')
42
>>> int('fortytwo')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'fortytwo'
>>>
```

If you want to catch an exception, use the `try-except` statement to enclose a block of code where a failure might occur. For example:

```
def spam(s):
   try:
      x = int(s)
      print('Value is', x)
   except ValueError as e:
      print('Failed: Reason %s' % e)
```

```
>>> spam('42')
Value is 42
>>> spam('fortytwo')
Failed: Reason invalid literal for int() with base 10: 'fortytwo'
>>>
```

Exceptions always have an associated value that is an instance of the exception type. The "`as e`" clause on the `except` captures this instance and puts it into a local variable `e`. If you print out the value, you'll usually just get the error message.

If you want to catch different kinds of errors, you can have multiple `except` blocks. For example:

```
try:
   ...
```

```
except ValueError as e:
    ...
except TypeError as e:
    ...
```

Or, if you want to group the exceptions together so that they're handled by the same except block, use a tuple:

```
try:
    ...
except (ValueError, TypeError) as e:
    ...
```

Certain functions in Python don't raise exceptions but rely on return codes instead. Such functions are rare, but one such example is the `find()` method of strings, which returns -1 if no match can be found:

```
>>> s = 'Hello World'
>>> s.find('Hell')
0
>>> s.find('Cruel')    # No match
-1
>>>
```

The end-of-file condition on files and sockets is also signaled by returning an empty string instead of raising an exception. For example:

```
>>> f = open('somefile.txt', 'r')
>>> data = f.read()      # Read all of the data
>>> f.read()            # Read at EOF
''
>>>
```

Again, such examples of using return codes are rare. Most of the time errors are indicated through exceptions, and catching an exception is a straightforward process using `try-except`.

How to Indicate an Error

In your own code, don't be shy about raising exceptions. If there is some kind of problem, use the `raise` statement to announce it:

```
def spam(x):
    if x < 0:
        raise ValueError('x must be >= 0')
    ...
```

A common mistake made by newcomers is to indicate errors in some other way. For example, with a `print` statement and maybe a special return code:

```
def spam(x):
    if x < 0:
        print('x must be >= 0')
        return None
    ...
```

There are all sorts of problems with such an approach. First, the output of the `print()` is easily lost or overlooked. Moreover, it's pretty likely that other code won't be expecting the `None` return code and won't check for it. Thus, the error might just disappear into the void. This can make for an interesting debugging session later. No, it is almost always better to loudly announce errors with the `raise` statement. That is the Python way—embrace it.

Another immediate problem with raising an exception concerns the exception type that you're supposed to use. Python predefines about two dozen built-in exception types that are always available (i.e., `NameError`, `ArithmeticError`, `IOError`, etc.). Most of these errors are most applicable to Python itself, but a few specific exceptions might be useful in application code. A `ValueError` is commonly used to indicate a bad value as shown. Raise a `TypeError` if you want to signal an error related to bad types (e.g., a list was expected, but the caller passed in a tuple). A generic `RuntimeError` is available to indicate other kinds of problems.

In larger applications, it may make more sense to define your own hierarchy of exceptions instead of relying on the built-ins. This is easily done using classes and inheritance. For example, you start by making a new top-level exception like this:

```
class MyError(Exception):
    pass
```

You can then make more specific kinds of errors that inherit from `MyError`:

```
class MyAuthenticationError(MyError):
    pass
class MyIntegrityError(MyError):
    pass
class MyTimeoutError(MyError):
    pass
class MyBadValueError(MyError):
    pass
```

Use the `raise` statement as before to indicate the exact error that you want:

```
def spam(x):
    if x < 0:
        raise MyBadValueError('x must be >= 0')
    ...
```

One advantage of defining your own hierarchy of exceptions is that you can more easily isolate errors originating from your application as opposed to those from Python itself or any third-party libraries you might have installed. You simply catch the top-level exception like this:

```
try:
    ...
except MyError as e:
    # Catches any exception that subclasses MyError
    ...
```

Isolating your exceptions can be a useful tactic for debugging. For example, if the code dies from a built-in exception, it might indicate a bug in your code, whereas code that dies due to one of your custom exceptions might indicate a bug in someone else's code (e.g., whoever is calling your code). By being more precise about exceptions, you can more easily assign blame when things go wrong—you want that.

What Exceptions Should You Catch?

Given that exceptions are the preferred way of indicating errors, what exceptions are you supposed to catch in your code anyway? It might seem counterintuitive, but I almost never write code to catch exceptions—instead, I simply let them propagate out, possibly causing the program to crash. As an example, consider this code fragment:

```
def parse_file(filename):
    f = open(filename)
    ...
```

Now, suppose that the user passes a bad filename and the `open()` function fails with an `IOError` exception. You could write the code to account for that possibility by wrapping the `open()` with a `try-except` like this:

```
def parse_file(filename):
    try:
        f = open(filename)
    except IOError as e:
        # Handle the error in some way  ???
        ...
    ...
```

However, if you do this, it suddenly raises all sorts of questions. For example, what are you supposed to do in the `except` block? Do you print a message? Do you raise an exception? If you raise an exception, how is it any different from `open()` raising an `IOError`? Last but not least, even if the code catches the error, is there any way that the function can proceed afterwards? If there is no file, there is nothing to parse. How would it work?

As a rule of thumb, you should probably never catch exceptions unless your code can sensibly recover and take action in some way. Just to illustrate, a much more likely scenario would be a parsing function that needed to account for bad values:

```
def parse_file(filename):
    f = open(filename)
    for line in f:
        fields = line.split()
        try:
            x = float(fields[0])
        except ValueError:
            x = float('nan')
        ...
```

Here, catching a possible exception and using it to take corrective action makes sense. These are the kinds of errors you should be concerned with—not errors for which there is no hope of sane recovery. Put another way, if something is going to fail spectacularly and there's no hope, it's often better to step back and let it fail. Don't let your code get mixed up in the middle of the mess.

Beware the Diaper Pattern

Now wait just a minute—surely I can't be advocating a coding style where you never catch errors. Python code might be running some kind of important service where it can't just crash and disappear with a traceback. It's important to catch the nuance of the previous section. Basically, you shouldn't be writing code that attempts to catch exceptions for which no recovery is possible *at that point*. The possibility of a sane recovery really depends on context. For example, a parsing function clearly can't continue if it can't read data. However, if that function was invoked from within a larger framework executing a request on behalf of a client in a distributed system, there might be code that broadly catches failures and reports them back to the client.

A common technique for broad exception handling is to enclose code in a `try-except` block like this:

```
try:
    ...
    statements
    ...
except Exception as e:       # Catch any error
    # Handle the failure
    ...
```

`Exception` is the root of all error-related exceptions in Python (note: certain exceptions such as `SystemExit` derive from `BaseException` and won't be caught here). Thus, this code will catch any programming error that might occur.

This is the so-called "diaper pattern" in action—code that catches anything. It's also one of the most dangerous exception-handling approaches to be using. Don't be the programmer that writes code like this:

```
try:
    ...
    statements
    ...
except Exception as e:
    print('Computer says no')
    return
```

Or worse yet:

```
try:
    ...
    statements
    ...
except Exception as e:
    # TODO: Whoops, it failed.
    pass
```

Such code is the fastest way to create an undebuggable program. Any failure whatsoever, including common programming errors such as a misspelled variable name, will simply result in a vague error message. You'll never be able to figure out what's wrong.

If you're going to catch all errors, it is imperative that you report the actual reason for the failure. At a minimum, you might do this:

```
try:
    ...
    statements
    ...
except Exception as e:
    print('Computer says no. Reason %s' % e)
    return
```

A much better alternative is to make the full traceback available somewhere. If it's not going to be reported directly to the end-user for some reason, it can be written to a log file using the `logging` module and the inclusion of the `exc_info=True` option to logging functions. For example:

```
import logging
log = logging.getLogger('mylog')

...
try:
    ...
    statements
    ...
except Exception as e:
    print('Computer says no.')
    log.error('FAILURE: %s', e, exc_info=True)
    return
```

Alternatively, you can produce the traceback message yourself and obtain it as a string using the `traceback` module. This can be useful if you want to do something with the traceback such as redirect it elsewhere (e.g., to an email message, a database, etc.). For example:

```
import traceback
try:
    ...
    statements
    ...
except Exception as e:
    print('Computer says no.')
    tb = traceback.format_exc()    # Create the traceback message
    # Do something with tb
    ...
    return
```

It should be noted that the traceback module has additional functions for pulling apart stack traces and formatting them. Consult the online documentation [1] for more information.

If your intent is to merely log the error while allowing it to propagate, you can use a bare `raise` statement to re-raise the exception. For example:

```
try:
    ...
except Exception as e:
    log.error('FAILURE: %s', e, exc_info=True)
    raise    # Reraise the exception
```

You, Yes You Did It!

As much as you might try to sanely handle errors in your code, dealing with errors in large systems is still tricky. One particularly nasty problem arises if you capture exceptions and then raise a different kind of exception to encapsulate the "failure" in some broad way. For example, consider the following code:

```
class OperationalError(Exception):
    pass

def run_function(func, *args, **kwargs):
    try:
        return func(*args, **kwargs)
    except Exception as e:
        raise OperationalError('Function failed. Reason %s' % e)
```

Now, watch what happens in Python 2:

```
>>> def add(x, y):
...     return x + y
...
>>> run_function(add, 2, 3)
5
```

```
>>> run_function(add, 2, '3')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 5, in run_function
__main__.OperationalError: Function failed. Reason
unsupported operand type(s) for +: 'int' and 'str'
>>>
```

Here, you get an error message and it has some details about the failure, but the information is woefully incomplete. In particular, the traceback contains no useful information about what actually happened in the `add()` function itself.

Chained exceptions [2] is one area where Python 3 shines. If you try this same code on Python 3, you get two tracebacks:

```
>>> run_function(add, 2, '3')
Traceback (most recent call last):
  File "<stdin>", line 3, in run_function
  File "<stdin>", line 2, in add
TypeError: unsupported operand type(s) for +: 'int' and 'str'

During handling of the above exception, another exception
occurred:

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 5, in run_function
__main__.OperationalError: Function failed. Reason
unsupported operand type(s) for +: 'int' and 'str'
>>>
```

This is exception chaining in action. You can further refine the exact error message by adding a `from` modifier to the `raise` statement like this:

```
class OperationalError(Exception):
    pass

def run_function(func, *args, **kwargs):
    try:
        return func(*args, **kwargs)
    except Exception as e:
        raise OperationalError('Function failed') from e
```

Now, the error message changes to the following:

```
>>> run_function(add, 2, '3')
Traceback (most recent call last):
  File "<stdin>", line 3, in run_function
  File "<stdin>", line 2, in add
TypeError: unsupported operand type(s) for +: 'int' and 'str'

The above exception was the direct cause of the following
exception:

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 5, in run_function
__main__.OperationalError: Function failed
>>>
```

In this case, you're seeing a chain of exceptions and information about causation. That's pretty nice.

Final Thoughts (and Advice)

Proper handling of errors is an important aspect of any application. However, you want to make sure you do it in a way that allows you to maintain your sanity. The following rules of thumb provide a summary of some of the ideas in this article:

- Prefer the use of exceptions to indicate errors. It is the most common Python style and will be less error prone than alternatives such as returning special codes from functions.
- Don't write code that catches exceptions from which no sensible recovery is possible. It's better to simply let the exception propagate to some other code that knows how to deal with the error.
- Be extremely careful when writing code that catches all errors. Make sure you always report diagnostic information somewhere where it can be found by developers. Otherwise, you'll quickly end up with undebuggable Python code.

As an aside, a recent article in *;login:* about catastrophic failures in distributed systems [3] reported that nearly 35% of these problems were caused by trivial mistakes in exception handling. Although not specific to Python, that article is definitely worth a read.

References

[1] https://docs.python.org/2/library/traceback.html (Traceback Module).

[2] https://www.python.org/dev/peps/pep-3134/ (Exception Chaining).

[3] D. Yuan et al., "Simple Testing Can Prevent Most Critical Failures: An Analysis of Production Failures in Distributed Data-Intensive Systems," *;login:*, vol. 40, no. 1, February 2015 (USENIX).

Donate Today: The USENIX Annual Fund

Many USENIX supporters have joined us in recognizing the importance of open access over the years. We are thrilled to see many more folks speaking out about this issue every day. If you also believe that research should remain open and available to all, you can help by making a donation to the USENIX Annual Fund at www.usenix.org/annual-fund.

With a tax-deductible donation to the USENIX Annual Fund, you can show that you value our Open Access Policy and all our programs that champion diversity and innovation.

The USENIX Annual Fund was created to supplement our annual budget so that our commitment to open access and our other good works programs can continue into the next generation. In addition to supporting open access, your donation to the Annual Fund will help support:

- USENIX Grant Program for Students and Underrepresented Groups
- Special Conference Pricing and Reduced Membership Dues for Students
- Women in Advanced Computing (WiAC) Summit and Networking Events
- Updating and Improving Our Digital Library

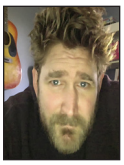
With your help, USENIX can continue to offer these programs—and expand our offerings—in support of the many communities within advanced computing that we are so happy to serve. Join us!

We extend our gratitude to everyone that has donated thus far, and to our USENIX and LISA SIG members; annual membership dues helped to allay a portion of the costs to establish our Open Access initiative.

www.usenix.org/annual-fund

iVoyeur
Graphios

DAVE JOSEPHSEN



Dave Josephsen is the sometime book-authoring developer evangelist at Librato.com. His continuing mission: to help engineers worldwide close the feedback loop. dave-usenix@skeptech.org

Hello again, intrepid reader. It seems like only yesterday that we were talking about implementing spread data in Graphite [1]. In that article I alluded to a tool called Graphios, with which you may not be familiar. More specifically, I was using it to collect metrics data from Nagios and inject it into Graphite, but I was a little short on the details surrounding that particular tool. This month I thought I'd correct that by taking some time to elucidate on Graphios.

Really, there are two very good reasons I want to talk about Graphios. The first is that although Graphios is not very widely used today (compared, to say, PNP4Nagios), it is certainly the easiest way to connect Nagios to systems like StatsD, Graphite, and Librato. Indeed, if you need to emit metric data from Nagios to one of the newer time-series analysis systems like InfluxDB or OpenTSDB, Graphios (via StatsD) is pretty much your only option besides coding up something yourself.

The second reason I want to talk about Graphios is that its creator Shawn Sterling and I recently spent the better part of several months ripping out its Graphite-specific backend and replacing it with a modular framework into which any sort of graphing system can be plugged. And I think you'll agree that tooting one's own horn is an excellent and time-honored reason to talk about anything in general.

As a result, Graphios works as a glue layer between Nagios and any Metrics System that supports the Carbon, StatsD, or Librato API protocols (which is to say, pretty much every metrics system today). As depicted in Figure 1, Graphios uses the `host_perfdata` and `service_perfdata` hooks (defined in your nagios.cfg) to read metric data from your perfdata log, and handles formatting and sending it to systems like Librato, StatsD, and collectd.

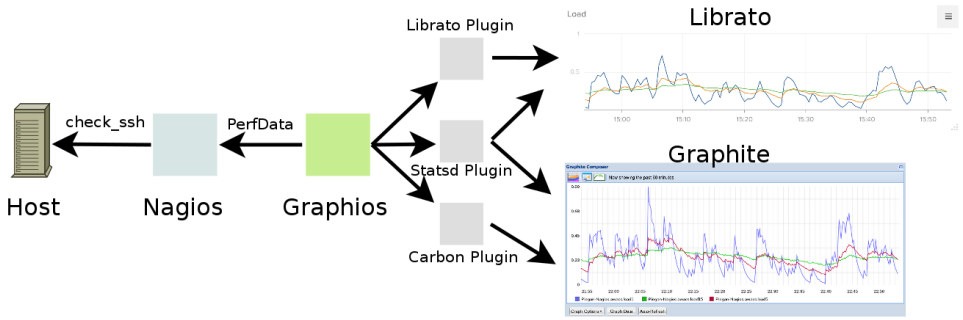


Figure 1: Graphios is a glue layer between Nagios and many different metrics systems.

Installation

Graphios is a Python program, so the easiest way to install it is with pip:

```
pip install graphios
```

It's also pretty easy to install Graphios manually. First, get the most recent version from Git with the following:

```
git clone https://github.com/shawn-sterling/graphios.git
```

Then copy the various files to the appropriate locations:

```
mkdir -p /etc/graphios
cp graphios*.py /usr/local/bin
cp graphios.cfg /etc/graphios/
```

Configuration Requirements

To get Graphios up and running, you'll need to manually configure three things:

- The Nagios config files that deal with host and service checks
- The nagios.cfg file
- The graphios.cfg file

If you use `pip install graphios`, the setup.py script will attempt to detect and automatically add a commented-out configuration to your nagios.cfg. The setup script does a pretty good job of this on all but the most bespoke Nagios setups (simply uncomment and restart Nagios), but given the configuration flexibility of Nagios, it's possible you'll need to manually intervene and modify the nagios.cfg yourself.

What's in a Name?

Nagios is a standalone monolithic system in that it assumes its check-command output will never be exported, that no system but Nagios will ever need to process it. So Nagios services generally have very simple names like PING or LOAD. In Nagios, it should be obvious to the operator what those names refer to because all of the context is inside the Nagios UI.

Graphing systems like Graphite, however, are not monolithic; they're designed to work alongside other monitoring systems and data collectors. Therefore they necessarily assume that all data is sourced externally (everything comes from some other monitoring system), and as a result they use dot-delineated, hierarchical metric names like Nagios.dc4.dbserver12.LOAD. In Graphite, a name like LOAD doesn't tell the operator anything about what system the metric refers to, much less how it was collected.

To be an effective glue layer, Graphios gives you a simple, transparent means to translate Nagios's simple, monolithic service names into context-rich hierarchical names that can be used by systems like Librato and Graphite. Specifically, Graphios can read metric prefixes and suffixes out of your Nagios service and host definitions using custom attributes. For example, a typical Nagios service description, excluding the minutiae normally packed into upper-level templates, looks like this:

```
define service{
  use                   generic-service
  hostname              box1,box2,box3
  service_description   SSH
  check_command         check_ssh }
```

The output of the `check_ssh` plugin looks like this:

```
SSH OK - OpenSSH_5.9p1 Debian-5ubuntu1 (protocol 2.0) |
time=0.009549s;;;0.000000;10.
```

Everything after the pipe is performance data [2]; these are the metrics Graphios exports. In this case, we have a single metric called time, which measures the response time of the SSH port (in this case, the SSH port responded to the check_ssh plugin in 0.009549 seconds). Graphios automatically prefixes the metric name with the hostname, so without doing anything at all, our metric name becomes:

```
box1.time
```

As I've already observed above, box1.time isn't a particularly meaningful metric name, so we can tell Graphios to put some additional context in front of this metric name by inserting a `_graphite prefix` custom attribute into the service definition like so:

```
define service{
  use                   generic-service
  hostname              box1,box2,box3
  service_description   SSH
  check_command         check_ssh
      _graphiteprefix           nagios.dc1 }
```

Graphios will now prepend this prefix to the metric name, making it:

```
nagios.dc1.box1.time
```

This is a little bit better, but we can insert some additional context about the service between the hostname and the metric name using a `_graphitepostfix` custom attribute in our service configuration like so:

```
define service{
  use                   generic-service
  hostname              box1,box2,box3
  service_description   SSH
  check_command         check_ssh
      _graphiteprefix         nagios.dc1
      _graphitepostfix                sshd.rt }
```

Graphios will now insert this string between the host and metric name, making it:

```
nagios.dc1.box1.sshd.rt.time
```

Now we have a pretty decent metric name for use with systems like Graphite and StatsD.

Configuring Nagios Perfdata Hooks

Next we need to configure Nagios to export performance data to a log file in a format that Graphios can understand. If you installed Graphios using `pip install graphios`, check the bottom of your nagios.cfg file for a block of configuration that begins:

```
# ###### AUTO-GENERATED GRAPHIOS CONFIGS
```

If you aren't already using Nagios perfdata hooks for something else, that is, if your currently running Nagios configuration *contains* `process_performance_data=0`, then you can simply uncomment this configuration block and restart Nagios.

If you're already using Nagios perfdata hooks for something like PNP4Nagios, or one of the other RRDtool-based graphing systems, chances are you can safely run both Graphios and your current tool set at the same time. Refer to the Graphios documentation [2] for instructions on how to set this up. You should also consult the Graphios setup docs if you don't see the auto-generated Graphios config at the bottom of your nagios.cfg, or if you didn't use pip to install.

Once you've configured Nagios to emit performance data, restart the Nagios daemon and verify that it's writing a log file to the Graphios spool directory (named by the service_perfdata_file attribute in your nagios.cfg) with a name like service-perfdata .1418637947. The file should contain lines that look like this:

```
DATATYPE::SERVICEPERFDATA  TIMET::1418637938  HOSTNAME::box1
SERVICEDESC::SSH  SERVICEPERFDATA::time=0.066863s;;;0.000000;10.000000
SERVICECHECKCOMMAND::check_ssh  HOSTSTATE::U  HOSTSTATETYPE::HARD
SERVICESTATE::OK  SERVICESTATETYPE::HARD  GRAPHITEPREFIX::nagios.dc1
      GRAPHITEPOSTFIX::sshd.rta
```

(Finally) Configure Graphios

Graphios installs its config file in /etc/graphios/graphios.cfg by default. This file is very well commented and, by and large, self-explanatory. There is a global configuration section and one section for each backend plugin that Graphios can write to. Plugins are generally enabled by setting their enable line to True and configuring the required attributes for the plugin. Here, for example. is a working configuration for Librato:

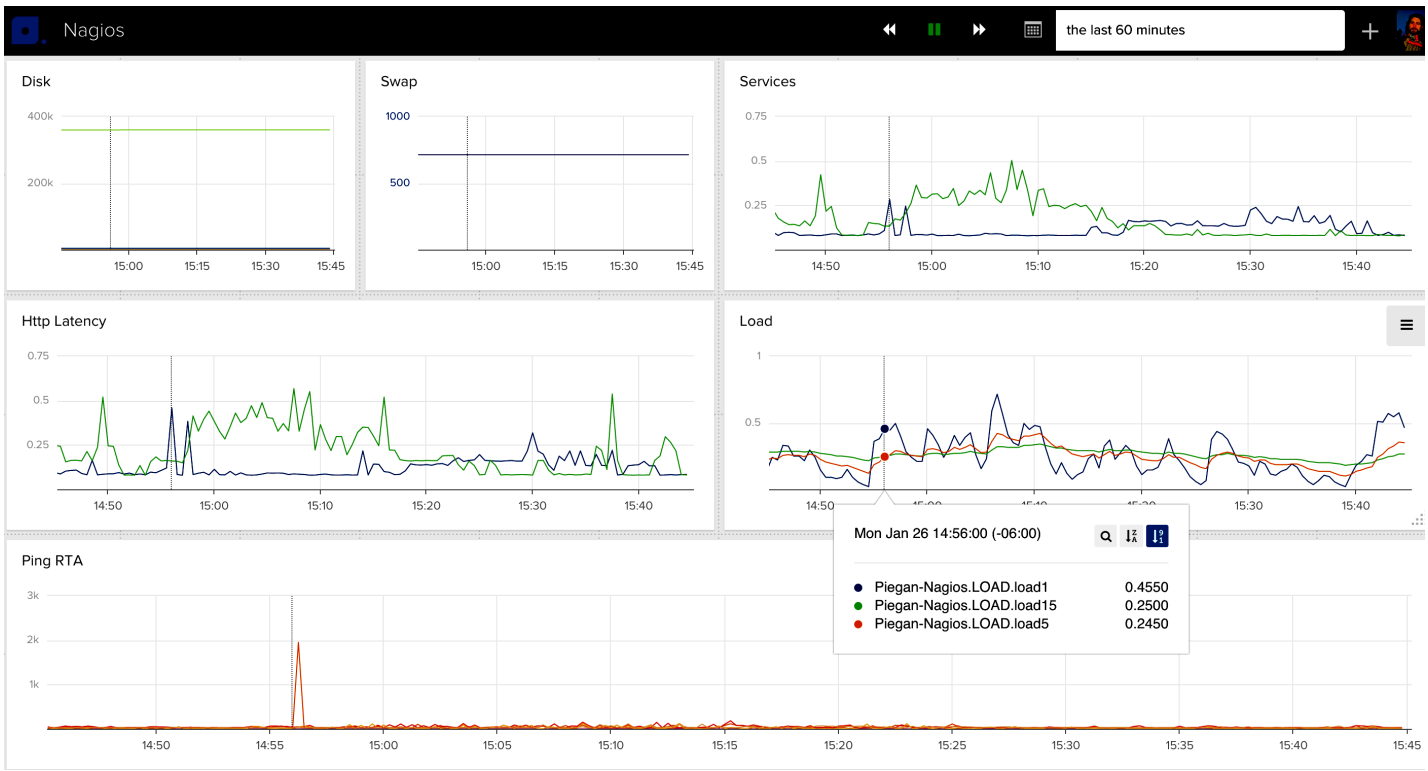


Figure 2: Lovely data, as if from heaven

```
enable_librato=True
librato_email = dave@librato.com
librato_token =
ecb79ff8a82areallylonggreatbigstringtokenything6b8cb77e8b5
librato_whitelist=["load","rta","swap"]
```

The whitelist attribute bears mentioning since, without it, Graphios would emit all performance data measured by Nagios to Librato, which could become expensive. As configured above, only metrics matching the regular expressions "load", "rta", and "swap" will be emitted to Librato. Here's a working configuration for StatsD:

```
enable_statsd=True
statsd_servers = 192.168.1.87:8125
```

You may enable multiple backend plugins (Librato AND StatsD) and even multiple comma-separated instances of the same backend plugin (four different StatsD servers and a carbon server), and Graphios will happily parse out and emit your Nagios Metrics to each backend system in turn. At this point you can run Graphios from the command line and see whether everything works as expected:

```
graphios.py --verbose
```

Now you should start seeing something like what's found in Figure 2, beautiful metrics data magically appearing in your metrics backend of choice.

Daemonizing Graphios

Graphios ships with init scripts for Debian and RPM-based systems, and these were installed automatically if you ran `pip install graphios` on a compatible system.

So How Does This Work Again?

Although its configuration necessarily borders on complex, Graphios is conceptually a very simple special-purpose log parser. It runs as a daemon, waking up on a configurable interval, checking for new performance data logs exported by Nagios, and processing them.

As I've already quite proudly mentioned, Graphios has a modular backend model that allows it to write to multiple metrics systems. When Graphios finds a new performance data file, it parses metrics out of it, computes appropriate metric names for the enabled backend plugins, and then it emits the metrics to each backend metrics system as required.

If you're running Nagios today, and you're still trapped in the RRDtool era, you owe it to yourself to install Graphios and experience the future of scalable metrics analysis systems like Graphite, InfluxDB, and OpenTSDB. One of the nicest features of Graphios for me has been its support for running multiple backends in parallel. Graphios makes it painless and simple to spin up and test new metrics systems, or combinations of metrics systems, without interrupting your production metric streams. I hope you find it as useful as I have.

Take it easy.

References

[1] Dave Josephsen, "iVoyeur: Spreading," *;login:*, vol. 40, no. 1, February 2015 (USENIX): https://www.usenix.org/publications/login/feb15/josephsen.

[2] https://github.com/shawn-sterling/graphios/blob/master/README.md.

For Good Measure
The Undiscovered

DAN GEER

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

Do not become the slave of your model.
—*Vincent Van Gogh*

Précis: Security metricians must steal all the techniques we can—we don't have the time to invent everything we need from scratch. This column does just that, motivated by the question of whether patching matters, a question that just will not go away.

A common problem in wildlife biology is simply this: "How many X are there in Y?" as in "How many frogs are there in the pond?" The most common method is "capture-recapture." The technique is simple and has been long applied not just to biology but also to things as disparate as how to adjust the US Census for undercount [1] to how many streetwalkers there are in Glasgow [2]. As with any statistical technique, there are assumptions, which we come to in a moment. First, this is the core process for estimating N, the number of frogs in the pond:

1. Take a sample of size n_1 and mark them.
2. Release the sample back into the wild.
3. Wait a short time, a week perhaps.
4. Take another sample of size n_2, counting the m_2 of the second sample that are found to be marked.
5. As $\frac{m_2}{n_2}$ should be the same as $\frac{n_1}{N}$, conclude $N = \frac{n_1 n_2}{m_2}$

That formulation is called the "Lincoln Index." As an example: catch 41 frogs and band them, then, a week later, catch 62 frogs and find that six are banded; we therefore estimate that there are

$$\frac{41 \times 62}{6} = 424$$

frogs in the pond. (Note: it is better to actually calculate $((n_1 + 1)(n_2 + 1)/(m_2 + 1)) - 1$ which yields an estimate of 377 frogs in the pond.)

The assumptions behind capture-recapture studies are that catching and marking the frogs does not change their behavior, that marked frogs completely mix into the pond's population, that any one frog, marked or not, is equally likely to be caught, that sampling is quick (preferably all at once), and that the population did not change between captures.

A second method, called "removal-capture," follows this process:

1. Catch n_1 frogs during a fixed-duration hunt and remove them.
2. Wait a short time, a week perhaps.
3. Catch n_2 frogs during a second fixed-duration hunt.

4. Calling N_0 the number of frogs on day 0, if

$$\frac{n_1}{N_0} = \frac{n_2}{N_0 - n_1} \text{ then } N_0 = \frac{{n_1}^2}{n_1 - n_2}$$

As an example: the first catch finds 78 frogs and the second 57; we therefore estimate that there were

$$\frac{78^2}{78 - 57} = 289$$

frogs in the pond on day 0.

The assumptions behind removal-capture studies are that the population is reasonably static, large enough for a significant catch in each subsequent sample yet small enough that a reduction in catch will be noticed, and that within a constant time interval a constant fraction of the frogs will be caught.

So why am I mentioning all this? In a May 2014 article in *The Atlantic* [3], Bruce Schneier asked a cogent, first-principles question: "Are vulnerabilities in software dense or sparse?" If they are sparse, then every vulnerability you find and fix meaningfully lowers the number of vulnerabilities that are extant. If they are dense, then finding and fixing one more is essentially irrelevant to security and a waste of the resources spent finding it. Six-take-away-one is a 15% improvement. Six-thousand-take-away-one has no detectable value. Eric Rescorla asked a similar question in 2004: "Is finding security holes a good idea?" [4] Rescorla established that it is a non-trivial question, as perhaps confirmed by our still not having The Answer.

In other words, we want to know how many frogs (vulnerabilities) there are in some pond (a software product). One might then ask whether either or both of the capture-recapture and removal-capture techniques might help us answer Schneier's and Rescorla's challenge, the challenge of picking a policy direction based on whether vulnerabilities are sparse or dense. Do we want a policy that skips patching in favor of rolling software fast enough to make it a moving target? [5] If we decide to keep patching, are we better off disclosing or keeping the repairs secret?

Starting at what may be the beginning, in a 1993 paper Vander Wiel & Votta [6] gave capture-recapture for software engineering a good airing. Their body of study was on latent errors of design in software projects and whether multiple, parallel design reviews might be structured so as not only to find design flaws but to also estimate how many further design flaws were as yet undiscovered. In other words, their frog is a design flaw and their pond is the design of a software project. The context of their work was an attempt to improve on what had been a quota system for design reviews at Bell Labs—a design reviewer had to find between a fixed minimum and a fixed maximum number of faults per page of the design document.

The Vander Wiel & Votta paper is worth a read if you want early statistical details. Their basic result was to assess how violating the assumptions (that are appropriate for wildlife biology) affected using the capture-recapture technique to estimate the number of design flaws in a software project. Quoting from their paper:

> Our approach treats the faults found by reviewers preparing for a design review as data from a capture-recapture sampling scheme. We use a Monte Carlo simulation to investigate the inaccuracies of the capture-recapture estimators due to assumption violations when faults have varying detection probabilities and reviewers have different capture probabilities. Although we would like to use data from real world design reviews to perform this study, it is impossible. We can not control the fault detection and reviewer capture probabilities in design reviews, nor could we ever hope to obtain the number of reviews required to perform a statistically significant study.

The problem at hand for security metricians is parallel—we cannot do controlled experiments, our vulnerability finders have broad ranges of skills (plus the most skilled ones are not talking), and vulnerabilities range from trivial to find to the kind of impossible that wins the "Underhanded C" [7] contest.

Their simulations enabled Vander Wiel & Votta to make some general recommendations for mitigating violations of the statistical assumptions. One is for when faults are not equally easy to find—we have that problem with respect to vulnerabilities, and they tell us that if we can group faults such that those within a group *are* equally easy to find, then we can do capture-recapture for individual groups of faults so long as the groups are large enough "that some faults in each group are discovered by more than 1 reviewer." Another is for when fault finders are not equally skilled. We have that problem, too, and they tell us that grouping fault finders by skill level might be worthy of study. (They did not pursue that option, but perhaps "we" should.)

The two decades since 1993 have seen a lot of experimentation with capture-recapture in the software engineering literature. Petersson et al. [8] reviewed that history, classifying the assumptions (and their violation) as:

> M_0 the probability of a fault being found is the same for all faults as is the ability of each inspector to find each fault
>
> M_h the probability of a fault being found is the same for all faults, but detection ability can vary from inspector to inspector
>
> M_t the probability of faults being found varies, but inspectors all have the same ability to find each fault

> M_{th} the probability of faults being found can vary and so can the ability to find a fault can vary from inspector to inspector

and, yes, each of these four needs a different modeling regime.

In any case, I am not aware of anyone approaching the issue of latent zero-day vulnerabilities, *per se,* with these techniques, techniques that software engineering has adapted from the biology world. Certainly, papers as early as Ozment & Schechter's "Milk or Wine: Does Software Security Improve with Age?" [9] looked at the declining rate of flaw finding within a software project under consistent management (OpenBSD), but that is subtly different and, in any case, should probably be evaluated as an example of removal-capture rather than an example of capture-recapture.

It seems to me that the most straightforward way to make a first quantitative effort here is to employ three or more independent penetration tests against the same target. Or have your software looked over by three or more firms offering static analysis. Scott & Wohlin's case study [10] with the KDE Open Source project and UIQ Technology might be worth copying.

Perhaps we can take a large body of code and look at the patches that have been issued against it over time. If you take a patch as a marker for a previously undiscovered flaw, then the rate at which patches issue is a removal-capture process. Were that process to maintain a relatively constant hum, then it might imply that software flaws are indeed dense—too dense to diminish with removals. Of course, patches for a commercial software system are not necessarily unitary—one apparent patch may actually fix several flaws. Rescorla concluded that fixing without disclosure is better than fixing with disclosure (and thus was "an advantage for closed source over open source"), but such a policy certainly doesn't help us do quantitative research with real data.

There is something here to work with for those who test or who can closely observe those who do. Be in touch; I'd like to work with you.

References

[1] Multiple articles in *Statistical Science*, vol. 9 no. 4, 1994.

[2] N. McKeganey et al., "Female Streetworking Prostitution and HIV Infection in Glasgow," *British Medical Journal* (1992), vol. 305, pp. 801–804.

[3] "Should U.S. Hackers Fix Cybersecurity Holes or Exploit Them?": www.theatlantic.com/technology/archive/2014/05/should-hackers-fix-cybersecurity-holes-or-exploit-them/371197.

[4] E. Rescorla, "Is Finding Security Holes a Good Idea?" Workshop on the Economics of Information Security, 2004.

[5] S. Clark, S. Collis, M. Blaze, and J. M. Smith, "Moving Target: Security and Rapid-Release in Firefox," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security* (ACM, 2014) pp. 1256–1266: dl.acm.org/citation.cfm?id=2660320.

[6] S. A. Vander Wiel and L. G. Votta, "Assessing Software Designs Using Capture-Recapture Methods," *IEEE Transactions on Software Engineering (1993)*, vol. 19, no. 11, pp. 1045–1054.

[7] http://www.underhanded-c.org/: "The goal of the contest is to write code that is as readable, clear, innocent and straightforward as possible, and yet it must fail to perform its apparent function. To be more specific, it should do something subtly evil."

[8] H. Petersson, T. Thelin, P. Runeson, and C. Wohlin, "Capture-Recapture in Software Inspections after 10 Years of Research," *Journal of Systems and Software* (July 2004), vol. 72, no. 2, pp. 249–264.

[9] A. Ozment and S. E. Schechter, "Milk or Wine: Does Software Security Improve with Age?" in *Proceedings of the 15th Conference on USENIX Security Symposium* (2006), pp. 93–104.

[10] H. Scott and C. Wohlin, "Capture-Recapture in Software Unit Testing—A Case Study," Blekinge Institute of Technology, 2004: www.wohlin.eu/esem08-1.pdf.

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.

Learn more at:
www.usenix.org/supporter

/dev/random
Distributed System Administration

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

Usually when we have an issue with multiple themes I pick one or the other to mock/expound upon/reveal my gross ignorance about, but on this occasion I've decided to combine them. I spent years as a system administrator, and, in common with most of you, what I laughingly refer to as my earnings are distributed amongst a wide variety of government agencies via taxation (and more taxation, and then some additional taxation after that in case there's anything left over), so I feel as qualified as anyone with similar qualifications to address both of this issue's topics.

Now, the problem with sysadmins is that they have not kept their image modern. By that I mean that while computing was evolving by leaps and bounds around them, the system administrator was stuck in 1998. I picked that year because that was probably the high point of my sysadmin career, if it can be said to have experienced a high point at all. It was shortly after that I wrote "Chasing the Wind" for *Security Focus*. When I look back at that fictional account now, it seems quaint: a relic of a bygone era. It was a product of a more innocent time and a more innocent author with a constant need to generate income. I'm not so innocent now. The sysadmin as a species hasn't changed much, either.

Before I proceed any further along the primrose path (pausing every so often to pluck out the primrose thorns), I want to take a brief side trip to that rising star of technological ubiquity, the much-vaunted Internet of Things. Most of the non-technical public, or so it seems to me, think of the IoT (when they think of it at all) as just a nifty way to control their refrigerator, air conditioner, cigar humidor, and basement sump pump via smartphone from the bar or airport (or airport bar).

But as my readers all know, IoT is far more than that. Once every macroscopic object in your household has its own wireless network interface and IP address, the possibilities for both utility and mayhem are virtually endless. But, uncharacteristically getting to the point, is the Internet of Things composed merely of things, or are traditional network devices included? Is your wireless router part of IoT, or just something to which the IoT connects? The reason this makes a difference is that nasty ol' bugbear of connectivity, security.

I had a dream recently (last night, actually, which as we've covered many times would be like back in the day for you), where the IoT played a major role. Instead of household appliances and smartphone-operated audiovisual devices, the network nodes were implanted in clones of "Things One and Two" from Dr. Seuss, "Thing" from the Addams Family, Ben Grimm, and an uncomfortably hideous grotesque humanoid from *The Thing*. It was one of my patented nocturnal excursions into self-induced insanity where I can't decide whether to laugh or scream in horror. This began when, as a child, I noticed that "laughter" is not only embedded in but constitutes almost 90 percent of "slaughter." That kind of thing takes its toll.

How is this relevant to security? My, aren't we the impatient one today? Other than childhood monsters that never left, the underlying trigger for this dream was my fear that, as with every other aspect of this worldwide meta-neural rete to which we are all now surgically

attached, security would be pasted onto the finished product with tape and paperclips, rather than designed into the fundamental infrastructure.

A few months ago I retired from many years of clubbing the US federal government over their pointy but durable heads—for the most part fruitlessly—with the hardwood 2x4 of infosec. I am, as happens to me frequently, simultaneously appalled and amused that cybersecurity is suddenly a major buzzword in the hallowed halls of executive and legislative governance. I and many others have been waving that brightly colored banner in their faces for two decades or more. They just patted us on the heads and said, "That's nice, little Jimmy: now run and play outside while the grownups drink their morning Kool-Aid."

Adrift on my tiny raft in a turgid sea of I-told-you-so, I find little comfort or triumph in the destination. I can't help but reflect on what a different world it might be had any of those elected talking heads actually listened to us, but there's really not much to be gained by what-ifs except weeds in the verdant lawn of the subconscious. Let's wend our way back to the titular topic now, shall we? Did anyone remember to leave breadcrumbs?

Distributed systems are all the rage these days, so it's only natural that we should also distribute the administration thereof, right? System Administration as a Service (SAaaS, which sounds like Frankenstein's monster before coffee) no doubt already exists in the cloud, although the idea that an unspecified collection of disbursed electronics and code could effectively be managed by an equally but separately disbursed unspecified collection of human neurons rather beggars the imagination. Of course, some are now calling for a total cessation to human intervention in this architecture, which proposal might well form the basis for my next terrifyingly jocular oneiric misadventure. Hopefully this time there will be snacks.

What would distributed system administration entail, precisely? Well, one relatively sane application I can see would be for systems that require around-the-clock (human) monitoring and tweaking. In that scenario the sysadmin duties could simply move along with the terminator (I'll be back in 24 hours). A somewhat trivial implementation, admittedly, but triviality has more or less been my hallmark as a humorist, so there you go.

Which brings me, willy-nilly, to my final point. Over the past nine years I have made a number of proposals in this space that ranged from the marginally sublime to the out-and-out ridiculous. That's because I am the, or rather a, humor columnist for this magazine. The number of readers who fail to pick up on that critically salient point is a source of continual amazement for me. I've been asked to present my, um, "engineering" ideas at some fairly high-profile conferences and meetings. I'm not sure whether this stems from my failure to be funny or just from the "nothing-is-too-stupid-to-try" entrepreneurial spirit of the interwebs these days. Most likely, both.

Regardless, if any of the systems or techniques or processes I propose seem as though they might be technically sound and implementable, I strongly encourage you to chart them out in your favorite graphics program or spreadsheet and then stare at them for a few minutes. I'm pretty sure the resulting representation of silly will scare you away, but if not and you decide to forge ahead to certain doom, please remember one thing: it's not my fault. You were warned.

Book Reviews

MARK LAMOURINE

Test-Driven Development with Python
Harry J. W. Percival
O'Reilly Media, Inc., 2014, 449 pages
ISBN 978-1-449-36482-3
Harry Percival is a convert to test-driven development (TDD) and he wants you to be one, too. Usually I have a real problem with this kind of enthusiast, but Percival's easy tone and a touch of quirky humor kept me reading.

Percival's chosen tools for writing about TDD are Python 3, Django, Git, and Selenium. He does assume experience coding and recommends several other texts for the reader who might not be ready. It would be good to brace yourself because it's quite a ride. At the end of the first chapter, he has the reader viewing a browser page popped up by Selenium from the first test page.

You might be forgiven if, after leafing through the body of the text, you thought that the book was really about Web development with Django. Percival touches on object relational mapping and persisting objects into a database in one chapter and on REST service behavior in the next. He treats Web forms and input validation and backend automation methods. When he comes to the content and behavior of the pages themselves, he includes some JavaScript (along with suggestions for testing). A closer look shows that he's applying TDD throughout the process. The application always drives the development (and hence the testing) activity.

In the final sections he covers advanced topics like mocking and continuous integration with Jenkins. Here the focus is back on the testing proper, and he closes by revisiting the Testing Goat. (Go ahead, Google it.)

With the breadth of tools and techniques here, Percival could fill a book much larger than this. Instead, he gives judicious references to other books and documents, inviting the reader to take a side trip and come back. He manages to instill the narrative with an invitation to try a new way of working while avoiding much of the preachiness that methodology books often have. If you're planning to learn or, better, to try TDD, this is a great place to start.

The Theoretical Minimum: What You Need to Know to Start Learning Physics
Leonard Susskind and George Hrabovsky
Basic Books, 2013, 238 pages
ISBN 978-0-465-0758-3
This isn't a programming or sysadmin book. It's a guide for the serious autodidact who is unsatisfied with typical whitewashed popularized books on physics. I'm presenting it here because its underlying premise, "the theoretical minimum," appeals to me.

The book is actually the result of a series of adult education courses that Susskind has run at Stanford University. It's the first of a series of serious physics courses for people who are curious nonprofessionals. The lectures are online and free at http://theoreticalminimum.com/courses. Susskind noted that his typical undergraduate students were often less than enthusiastic about actually learning physics. By contrast, the students who came to the adult ed courses were all motivated by their personal curiosity. Teaching them was more fun.

Susskind's idea is to pare down the ideas and the mathematics of classical physics so that an intelligent nonacademic can understand them in the terms a professional would recognize. This means glossing over many of the side routes and much of the theoretical depth that would be included in a two- or three-semester undergraduate course in classical physics. At the same time, he doesn't skimp on the depth and rigor in his discussion of the ideas he presents.

In the course of eleven chapters (corresponding to the eleven lectures of the classroom course), Susskind presents the theory and mathematics of classical physics. The first chapter would be familiar to anyone who has taken a high-school course: vectors. In the second chapter, the reader has to tackle the concepts of integral calculus and the symbolic language needed to express them. Susskind progresses rapidly through dynamics, energy, and one of the best discussions of the relationship between symmetry and conservation that I've read.

One of the things I like most about this book is that the mathematics is explained in terms of the physics and vice versa. I've always had difficulty understanding why college math and physics departments insist on treating the two topics separately. The mathematicians seem not to want to sully themselves with physics, and the physicists insist they can't begin work unless the students come in already fluent in calculus. When the two are combined, they produce a unified comprehensible whole that is impossible to capture separately.

I'm reviewing this book here for two reasons. The first is that many of us are inveterate geeks who love learning for its own sake, and if you're one of those, The Theoretical Minimum is an excellent read. The other is a proposition to the community of coders and system administrators: Is this something we can do? Is it possible to strip away the trivia and minutiae that each of us holds dear and leave something useful? What would The Theoretical Minimum mean for us?

REAL SOLUTIONS FOR REAL NETWORKS

FREE CD or DVD in Every Issue!

Each issue delivers technical solutions to the real-world problems you face every day.

Learn the latest techniques for better:

- network security
- system management
- troubleshooting
- performance tuning
- virtualization
- cloud computing

on Windows, Linux, Solaris, and popular varieties of Unix.

6 issues per year!

ORDER ONLINE AT: shop.linuxnewmedia.com

NOTES

The State of the USENIX

by Casey Henderson, USENIX Executive Director

I'm writing this just over a year after taking the USENIX helm solo as Executive Director, after 12 years as a USENIX staff member. It's been an exciting time of rapid change and growth, building on the foundation that Anne Dickison and I constructed as Co-Executive Directors. Although I interact with many of you at our events and we provide some snippets in our newsletter, I'd like to take the opportunity that our magazine, *;login:*, gives us to go into more detail about where USENIX is today. Although it's been a challenging ride following the recession, I'm delighted to report that USENIX is climbing onto solid ground both financially and programmatically.

USENIX's 40th anniversary arrives at a time when our year-end finances for 2014 are in the black for the first time since 2008. USENIX is fortunate to have a reserve fund built on generous donations, which allowed us to survive these past few years. While it has been disheartening for those of us on the staff and on the Board of Directors to withdraw on average half a million dollars a year to stay afloat during the aftermath of the 2008 financial crisis, we are proud to have stabilized this situation.

Our excellent investment portfolio management team helped minimize losses as the market took deep dives, and we have kept a tight rein on our financial commitments. It's a challenging way to live, although it's a better situation than that of many other non-profit organizations that exist without the cushion of a reserve fund. This position provided the optimal time to question what was truly essential to our mission, what should be jettisoned, and what should be pursued not only in the name of survival, but also in the name of growth. I personally didn't see much point in USENIX emerging from a financial crisis in a similar position to the way it went in. Thus rather than doing things as we've always done, we are now actively honoring the agility of our own technology-centric community to bend and flex with new ideas and information.

When I began working at USENIX in 2002, we were in the process of establishing some of the conferences for which we're now celebrated—for example, FAST and NSDI—and once again, we are taking similar leaps to build major events for emerging communities. As gratifying as it is to see these well-established conferences celebrate Test of Time Awards to showcase the work that was presented at conferences where I first updated the USENIX Web site, it's been even cooler to begin again to create new events to respond to trends in the field. The inaugural SREcon, focused on Site Reliability Engineering, sold out last year and is set to be even larger this year. We've created a sister event in Europe, and the talk proposals are flooding in. We have more new events in the pipeline that I look forward to sharing with you soon.

USENIX excels at running conferences logistically so that our volunteers can focus on the most important aspect: the programs. We now offer our event services, such as publishing open access proceedings, to other conferences. We've also become a grantee of the National Science Foundation, providing logistics for the Secure and Trustworthy Cyberspace (SaTC) Principal Investigators Meeting in January and the Big Data Regional Innovation Hubs in April. These events fit within the USENIX mission and also serve to broaden the scope of our community.

FAST has celebrated the highest attendance ever for the past two years. Bridging the gap between industry and academia has always been the key to its success, a feature that we're trying to emulate in other conferences more effectively.

NSDI will have two major co-locations in 2015 and 2016, with the ACM/IEEE Symposium on Architectures for Networking and Communications Systems this May and with the Open Networking Summit and the ACM SIGCOMM Symposium on SDN Research in 2016. These complementary events will enhance the existing programs while maintaining NSDI's focus on addressing research challenges within the networked systems field.

2015 marks the first USENIX Annual Technical Conference that isn't wrapped in a Federated Conferences Week in several years. The FCW concept of gathering multiple events together and allowing attendees to float among them, cross-pollinating in the hallway track, was a great idea but, in practice, it didn't draw enough attendees and drained precious resources. USENIX ATC '15 doesn't stand alone; HotCloud and HotStorage will keep it company and enrich the week.

We received a record-high 430 USENIX Security complete paper submissions for 2015—a significant jump from last year, which was also the highest-ever at that time. The papers portion of the program has grown so much that we now have two tracks of refereed papers and will expand to three tracks in order to keep invited talks as part of the program. We partnered with Facebook to establish the Internet Defense Prize in 2014, driving more interest in participating in the event.

OSDI in 2014 was the largest in history, with 584 attendees. It continued our recent trend of shorter paper presentations, which allows us to accept more papers to publish. It's a good example of our not doing things the way we've always done them—I can't tell you how many programs I've curated with

three papers per 90-minute session—and it's delightful to work with program chairs who approach program construction with creativity.

LISA13 and LISA14 marked our first major restructuring of the conference in a decade. The system administration field is in the midst of a sea change. DevOps and Site Reliability Engineering are emerging while managing systems at scale is now as much about the software and organizational collaboration as it ever was about the hardware. Many conferences serve this segment of our community, but ours is the original, the one created by sysadmins for sysadmins, and it's important to USENIX that LISA serve them well as a must-attend venue. We began to refocus the content and to weave learning and practice together in 2013. Building on that momentum, LISA14 was the largest and liveliest in many years. LISA15 will build on their success. Help us drive the field and the conference forward: submit a proposal for the program by April 17.

There is much more on our to-do list. USENIX is a membership organization in an era when fewer people join associations. We are a nonprofit that seeks the support of its constituents via its Annual Fund, but understands that folks are already paying dues and may not even realize that we are a non-profit. We are proud to be the only major computing association to offer truly open access publications, but are challenged by our business model, which is not designed to support the costs associated with free access. We have had to reduce the number of Good Works projects we support and we must determine how to move toward restored funding. We've established our Women in Advanced Computing (WiAC) initiative, but need to hear more from the community about how to be most effective in this arena. We have more students interested in attending our conferences than we have grant money to dispense. We ourselves have a hard time describing everything we do here at USENIX in one breath, which makes it difficult to tell people why we matter. We have come this far, though, and we are finally in a place where we genuinely believe that we will conquer these challenges—and move forward with the support of our members and the broader community.

Thanks to my colleagues on the Board, the staff, and our hundreds of volunteers for partnering with me on what has been a monumental effort to reinvent USENIX while remaining true to our roots. In particular, I thank past and current USENIX presidents Clem Cole, Margo Seltzer, and Brian Noble for pushing the boundaries of what we thought we could achieve and never giving up on this amazing organization. I look forward to working with all of you in the coming years to help USENIX reach a point of absolute stability and constant growth.

USENIX Board of Directors

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

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

VICE PRESIDENT
John Arrasjid, *EMC*
johna@usenix.org

SECRETARY
Carolyn Rowland, *National Institute of Standards and Technology*
carolyn@usenix.org

TREASURER
Kurt Opsahl, *Electronic Frontier Foundation*
kurt@usenix.org

DIRECTORS
Cat Allman, *Google*
cat@usenix.org

David N. Blank-Edelman, *Apcera*
dnb@usenix.org

Daniel V. Klein, *Google*
dan.klein@usenix.org

Hakim Weatherspoon, *Cornell University*
hakim@usenix.org

EXECUTIVE DIRECTOR
Casey Henderson
casey@usenix.org

USENIX Member Benefits

Members of the USENIX Association receive the following benefits:

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

Access to *;login:* online from December 1997 to the current month: www.usenix.org/publications/login/

Access to videos from USENIX events in the first six months after the event: www.usenix.org/publications/multimedia/

Discounts on registration fees for all USENIX conferences.

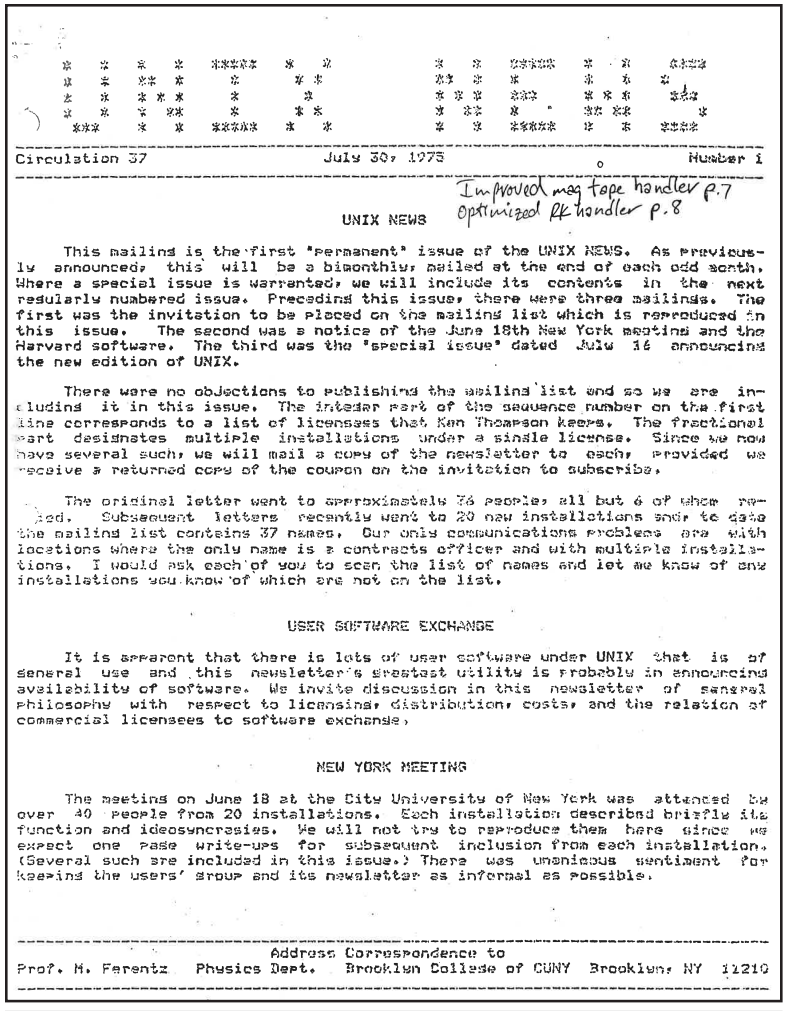
Special discounts on a variety of products, books, software, and periodicals: www.usenix.org/member-services/discount-instructions

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

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

Introducing *UNIX News*

The first issue of *UNIX News* was published by Mel Ferentz of Brooklyn College in July 1975. Two years later, *UNIX News* became *;login:* the UNIX Newsletter. To celebrate our 40th anniversary, we are reprinting some issues of *UNIX News* and early *;login:* newsletters. We are also reprinting some of the more popular articles that have appeared in *;login:* the USENIX magazine.



Circulation 37 July 30, 1975 Number 1

Improved mag tape handler p.7
Optimized RK handler p.8

UNIX NEWS

This mailing is the first "permanent" issue of the UNIX NEWS. As previously announced, this will be a bimonthly, mailed at the end of each odd month. Where a special issue is warranted, we will include its contents in the next regularly numbered issue. Preceding this issue, there were three mailings. The first was the invitation to be placed on the mailing list which is reproduced in this issue. The second was a notice of the June 18th New York meeting and the Harvard software. The third was the "special issue" dated July 16 announcing the new edition of UNIX.

There were no objections to publishing the mailing list and so we are including it in this issue. The integer part of the sequence number on the first line corresponds to a list of licenses that Ken Thompson keeps. The fractional part designates multiple installations under a single license. Since we now have several such, we will mail a copy of the newsletter to each, provided we receive a returned copy of the coupon on the invitation to subscribe.

The original letter went to approximately 76 people, all but 6 of whom responded. Subsequent letters recently went to 20 new installations and, to date the mailing list contains 37 names. Our only communications problems are with locations where the only name is a contracts officer and with multiple installations. I would ask each of you to scan the list of names and let me know of any installations you know of which are not on the list.

USER SOFTWARE EXCHANGE

It is apparent that there is lots of user software under UNIX that is of general use and this newsletter's greatest utility is probably in announcing availability of software. We invite discussion in this newsletter of general philosophy with respect to licensing, distribution, costs, and the relation of commercial licensees to software exchange.

NEW YORK MEETING

The meeting on June 18 at the City University of New York was attended by over 40 people from 20 installations. Each installation described briefly its function and ideosyncrasies. We will not try to reproduce them here since we expect one page write-ups for subsequent inclusion from each installation. (Several such are included in this issue.) There was unanimous sentiment for keeping the users' group and its newsletter as informal as possible.

Address Correspondence to
Prof. M. Ferentz Physics Dept. Brooklyn College of CUNY Brooklyn, NY 11210

The first issue of *UNIX News*. We have reproduced the text exactly as it appeared in the original, including any typographic errors. Note from the editor: "Printed on an LA36 from NROFF directly to Hectomaster. Great fun! MF"

UNIX News, July 20, 1975

This mailing is the first "permanent" issue of the UNIX NEWS. As previously announced, this will be a bimonthly, mailed at the end of each odd month. Where a special issue is warranted, we will include its contents in the next regularly numbered issue. Preceding this issue, there were three mailings. The first was the invitation to be placed on the mailing list which is reproduced in this issue. The second was a notice of the June 18th New York meeting and the Harvard software. The third was the "special issue" dated July 16 announcing the new edition of UNIX.

There were no objections to publishing the mailing list and so we are including it in this issue. The integer part of the sequence number on the first line corresponds to a list of licenses that Ken Thompson keeps. The fractional part designates multiple installations under a single license. Since we now have several such, we will mail a copy of the newsletter to each, provided we receive a returned copy of the coupon on the invitation to subscribe.

The original letter went to approximately 76 people, all but 6 of whom responded. Subsequent letters recently went to 20 new installations and, to date the mailing list contains 37 names. Our only communications problems are with locations where the only name is a contracts officer and with multiple installations. I would ask each of you to scan the list of names and let me know of any installations you know of which are not on the list.

User Software Exchange

It is apparent that there is lots of user software under UNIX that is of general use and this newsletter's greatest utility is probably in announcing availability of software. We invite discussion in this newsletter of general philosophy with respect to licensing, distribution, costs, and the relation of commercial licenses to software exchange.

New York Meeting

The meeting on June 18 at the City University of New York was attended by over 40 people from 20 installations. Each installation described briefly the function and idiosyncrasies. We will not try to reproduce them here since we expect one page write-ups for subsequent inclusion from each installation. (Several such are included in this issue.) There was unanimous sentiment for keeping the users' group and its newsletter as informal as possible.

The next meeting in the East will be October 6 at the City University of New York and the following meeting in early spring at Harvard. By October there should be considerable

experience with the new system, and by spring general experience with the Harvard system.

Ken Thompson described some of the features of the new system and some benchmarks run on the 11/70. He estimates the raw cpu gives a factor of 2.5 improvement in performance for UNIX, and that with the new peripherals the factor is about 3.0.

New System Available

The Sixth Edition—June 1975 of the UNIX system is now available for distribution to licensees. Commercial users should contact Western Electric for details. Academics can receive the new system for a service fee of $150.00. Normal distribution is on 800 bpi - 9 track tape. You need not send a tape. Just a check for $150.00 made out to Bell Laboratories, Inc. and sent to:

I. B. Biren, Room 2c-548
Bell Laboratories, Inc.
Computing Information Services Group
Murray Hill, NJ 07974

The tape contains a single file which extracts to 3 RK-packs or equivalent. These contain:

Pack 0	The system except for /usr/source
Pack 1	/usr/source
Pack 2	Documentation in machine readable form

Those who require distribution on RK-packs should send two or three packs along with their checks. The package also includes one hard-copy of each of the 19 documents.

Among the new "goodies" are:

1. Separate I and II space for the resident monitor on 11/45s and 11/70s
2. Huge files (up to 16 megabytes)
3. A preprocessor for structured Fortran
4. TMG
5. A preprocessor for DC, with arbitrary precision
6. Many fixes and rewrites of system programs from "as" to "c"
7. Much improved comments embedded in system source
8. More graceful death on running out of resources and other crashes

Other Software Available

The MUNIX paper which starts on page 4 announces the availability of their system. I have a recent note from professor Allen saying he expects to have it available in the very near future.

Harvard has announced the availability in the near future of their software. It will be available to other academic institutions for the nominal cost of reproducing it. The system is running in a heavy-use student environment and they expect to have some documentation by the end of the summer. For details write:

Lewis A. Law
Director of Technical Services
Science Center, Harvard University
1 Oxford Street
Cambridge, Mass. 02138

Requests for Software

From P. De Souza, Heriot-Watt University:

We are interested in getting in touch with UNIX users who may have developed a BCPL compiler/interpreter, a driver for Vector General display, or a software link to a PDP-10.

Installation Descriptions

University of Saskatchewan

PDP11/40 with 40Kw of core (expanding to 64Kw)
3 terminals (2 more on order)
1 DC11 dial-up interface and a CDI Teleterm 1030
2 RK11 disk drives (1 on order)
1 DH11 on order to replace current line interfaces

We also have a PDP11/20 with TTY, high speed paper tape and a VT01 display scope. This is currently connecting to the PDP11/40 by a DL11-E serial line but will soon be replaced by a DR11-C parallel interface. One current project is to write a monitor for the PDP11/20 so that its peripherals become available to UNIX users.

MUNIX—A Multiprocessor UNIX

B. E. Allen and G. L. Barksdale, Jr.
Computer Science Group, Naval Postgraduate School
Monterey, California 93940

The Naval Postgraduate School Signal Processing and Display Laboratory is a university laboratory engaged in research efforts in computer graphics, signal processing, operating systems, and hybrid computing. The laboratory is used for student instruction as well as for student and faculty research.

The configuration of the Signal Processing and Display Laboratory is shown in Figure 1. The system can be viewed as a three bus ensemble, with the respective functions of data acquisition, signal processing, and display. When bus cycles are not required by real-time processes, the data acquisition and display busses support program development activities. The display system includes a 256K word fixed head disk, a Ramtek color display, a Tektronix 4014 display with enhanced graphics, a Vector General 3D system, a Hughes Conographic console, a data tablet, a Versatek printer/plotter, and an EPC graphic recorder. Peripherals for the Data Acquisition controller include both large (96M words) and small (2.5M words) disk systems, magnetic tapes, a card reader, a line printer, and a sixteen line programmable terminal multiplexer. Dual ported core memory (88K words) is accessible from either UNIBUS. The signal processing subsystem consists of a CSP 125 controller with 4K words of 125 nanosecond memory, an array processor, and two 16K word banks of three ported memory. UNIX compatible device drivers have been developed for each of these peripherals.

To control this diverse hardware suite, we have evolved MUNIX, a tightly-coupled symmetric multiprocessor version of UNIX. A single copy of the system residing in shared memory is executed by both processors independently. P and V operators are used for synchronization. In order to provide the increased address space necessary to support the multiprocessor system, UNIX was modified to separate kernel I and D space. In support of the signal acquisition research, a new process classification, real-time, has been added. When a process is granted real-time status, it is locked in memory, given the highest priority possible, and preemptively allocated a processor whenever it comes ready.

Other completed work includes the development of a dynamic symbolic debugging tool having breakpoint capability, a rather basic PDP II virtual machine monitor which executes under MUNIX, several on-line diagnostic packages, a line editor which facilitates correction of typing mistakes, system calls which gracefully stop or bootstrap the system, and enhancements to the text editor, the text processor, the C compiler, and the loader. Work presently underway includes a performance measurement subsystem, several adaptive schedulers, a demand paged memory manager, and a hardened file system.

NPS developed software is available as a nine track tp tape to any Bell Labs approved site.

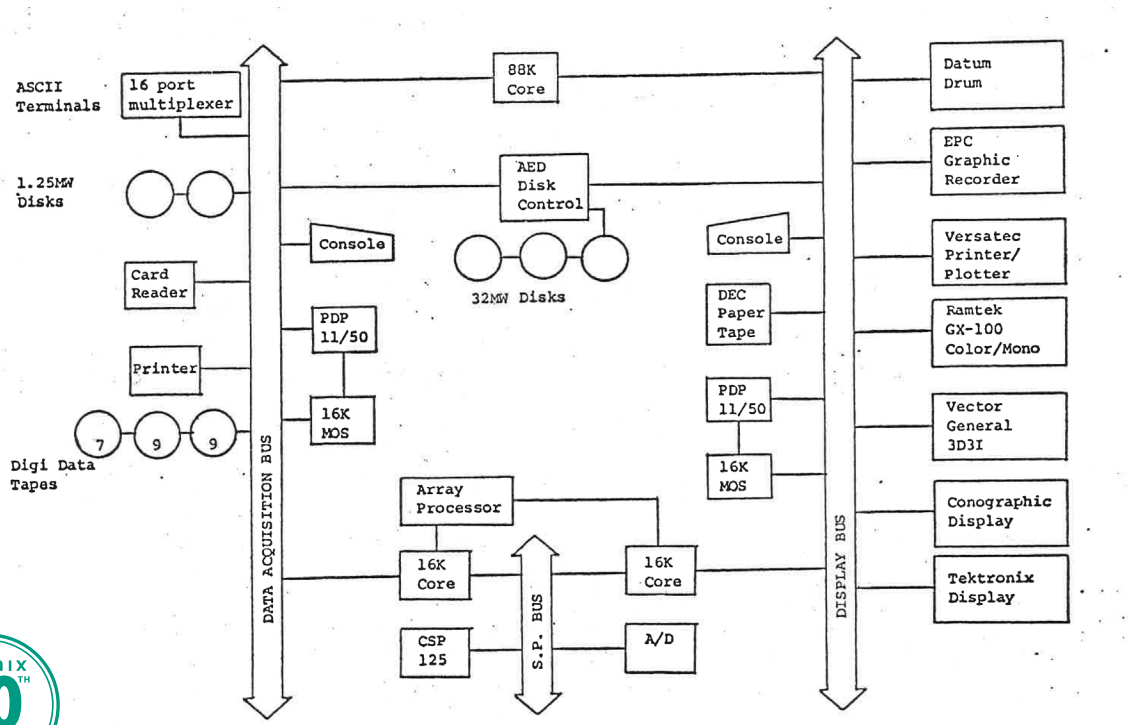


Figure 1. Configuration of the Signal Processing and Display Laboratory

Toronto UNIX System

1) Hardware

Device	Existence	Driver
a) PDP11/45 – floating point – 80+K core	Yes	
b) SI 9500-I Disk	Soon	No
c) Diva dd14 Disk	Soon	No
d) 3-Rivers Graphic Wonder	Yes	Yes and No
e) GT-40	Leaving Soon	Yes
f) Versateck D1200A Printer/Plotter - DMA	Soon	No
g) Colour Video System	Being Built	No
h) Summagraphic Data Tablet	Yes	Yes
i) Calcomp Microfilm Plotter	Yes	No
j) line printer	Yes	Yes
k) card reader	Yes	No
l) 1600 BPI tape drive	Yes	Yes

2) Software Already Developed

a) GT-40 driver

b) New improved mag tape driver
– allows seeks in raw mode
– knows about files
– crashes less frequently

c) C paragrapher

d) "grabcore" – a system routine to free up and reserve a specific piece of core for double-port devices

Boston Children's Museum

UNIX at the Children's Museum has been fully operational since August, 1974. Development work jointly with Harvard University began the previous winter, making us one of the first non-Bell users.

Our hardware configuration includes:

* PDP11/40 processor with EIS
* 48K core memory (MM/MF11-L)
* KW11L line clock
* 2 RK03 (a.k.a. Diablo) disk drives on RK11-c controller
* 6 VT05 terminals operating at 600 and 2400 baud on DL11-E controllers
* 1 LA30 DECwriter at 300 baud on DL11-A
* 1 ASR33 teletype on DL11-A
* 1 ComData modem on dialup line, 110 baud on DL11-E
* 1 LP11-HA upper/lower case 60-column line printer
* 1 VOTRAX VS-5 voice synthesizer on DL11-E
* 1 QUME G30 high-quality 30-cps printer (a.k.a. Diablo HyType, or the guts of the GSI etc. terminal) on DR11-C

Further, we are designing and will begin construction soon on several new hardware devices and interfaces, including a dirt-cheap DR11-C equivalent that is capable of driving our scaled-down elcheapo versions of things like the LOGO project's "turtle."

Our hardware and software is extensively kidproofed, and modifications have been made to the UNIX terminal driver to include modes whereby newline characters are ignored on "empty" or "null" lines, and whereby all characters typed by the user are thrown away if the system is in the midst of typing on the terminal. Attractive rubout handling (backspace-erase line) has also been added for VT05 terminals.

Software that we have developed that may be of interest to others includes:

* FOCAL, written in C and modeled after PDP-8 FOCAL by a high-school student
* A PDP-8 simulator (simple memory-and-a-single-terminal machines only at this time), also in C, by the same student (interrupts are not currently being supported but are being worked on)
* Rewritten standard UNIX shell (pipelines not yet implemented) with user-settable prompts, a "change to default directory" command, standard accounting options, a monitor option that copies all typein to a hidden file (for keeping tabs on potentially malicious users), and others
* A new more-conversational PS command that displays critical process data in English (SWAPPED/IN CORE, SLEEP/WAIT/RUN, etc.)
* An RK disk driver that optimizes seeking through queue-diddling

Under development and scheduled for imminent completion is a general-purpose information storage and retrieval system. A license fee will probably be made for this package, but all of the other items listed above are available free to nonprofits on request. Please contact me to discuss media conversion: we can supply RK disk, DECtape, or paper (sak) tape.

Bill Mahew, The Children's Museum, Jamaicaway, Boston, MA

Do you have a USENIX Representative on your university or college 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, the Campus Representative receives access to the members-only areas of the USENIX Web site, free conference registration once a year (after one full year of service as a Campus Representative), and electronic conference proceedings for downloading onto your campus server so that all students, staff, and faculty have access.

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

Dueling UNIXes and the UNIX Wars

PETER H. SALUS

Peter H. Salus is the author of *A Quarter Century of UNIX* (1994), *Casting the Net* (1995), and *The Daemon, the Gnu and the Penguin* (2008).
peter@pedant.com

For nearly a decade, UNIX was UNIX, the operating system from Bell Labs.

By 1979, we had PWB 2.0 (Programmer's Workbench), Version 7, 32V (the port of V7 to the VAX architecture), and 3BSD (the first Berkeley release for the VAX). UNIX, after all, was not an AT&T OS. It was a "telecommunications support tool." This was a result of the 1956 "consent decree," which enjoined AT&T/Western Electric from "commencing...manufacture for sale or lease any equipment" other than that used in telephony or telegraphy. One of the few exceptions permitted to AT&T was "[experimentation] for the purpose of testing or developing new common carrier communications services"—UNIX.

UNIX was worldwide at the end of the '70s: Australia in 1974; the UK in 1973; the Netherlands, Belgium, Austria, Israel, Japan, the US, and Canada had picked up the "new" system. And there were also commercial UNIX companies. The earliest was Whitesmiths, founded in 1978 by P. J. ("Bill") Plauger, and then the Wollongong Group in Australia and HCR in Toronto. In 1979, Microsoft and the Santa Cruz Operation brought out XENIX2 and Berkeley produced 4BSD, each a V7 derivative.

By the early '80s, UniSoft had released UniPlus+; mt Xinu (UNIX tm backwards) had been founded; and Apollo, DEC, Eakins, Gould, Integrated Solutions, Masscomp, NSC, and Wollongong were also marketing Berkeley UNIX. System III or System V derivatives were being marketed by AT&T, Altos, Apollo, Compaq, Convergent, HP, Honeywell, IBM, ITT, Intel, Interactive, Masscomp, Microport, Microsoft, Motorola, NCR, NUXI, Opus, SCO, Silicon Graphics, Sperry, Sun, Tandy, UniSoft, and Wollongong. Finally, a host of vendors, including Amdahl, Apple, Cray, DEC, Data General, HP, IBM, and Motorola, offered proprietary versions of UNIX, some based on 4.1 or 4.2BSD.

UNIX, which began in 1969 quite a bit smaller than MS-DOS, had become obese by its 18th birthday and was drowning in alphabet soup.

At the June 1986 USENIX conference in Atlanta, many AT&T staff wore buttons which read, "System V: Consider it Standard," and a number of major vendors were promoting products based on System V. On the other hand, System V did not yet have TCP/IP networking built in and BSD 4.2 did; vendors of engineering workstations were nearly all using BSD, and buttons and posters that said "4.2 > V" were available. (I still have mine.)

In late 1987, AT&T announced that it had purchased a large percentage of Sun Microsystems and that Sun would receive preferential treatment as AT&T/UNIX Systems Labs developed new software. Sun announced that its next system would not be a further extension of SunOS (which had been based on Berkeley UNIX) but would be derived from AT&T's System V, Revision 4. A shiver ran through the UNIX world: the scientific community felt that Sun was turning its back on them, and the other vendors felt that the "special arrangement" would mean that Sun would get the jump on them.

DEC, in particular, sensed that AT&T was no longer the benign, benevolent progenitor of UNIX.

The direct result of this was a meeting at DEC's Western offices in Palo Alto, on January 7, 1988. Participants present represented Apollo, DEC, Gould, Hewlett-Packard, Honeywell-Bull, InfoCorp, MIPS, NCR, Silicon Graphics, UniSoft, Unisys, and a few others. Because the offices were at 100 Hamilton Avenue, the attendees were referred to as the Hamilton Group. On January 15, the Group sent a telegram to James E. Olson, CEO of AT&T, requesting a meeting with Vittorio Cassoni, Senior VP of AT&T's Data Systems Division, during the week of January 25, 1988.

(UniForum and USENIX both met in Washington, D.C. that week, which culminated in the "second Washington snowstorm.")

Larry Lytle of HP called a preliminary meeting at the JFK Marriott for the evening of Wednesday, the 27th. The meeting with Cassoni was held the next day. Where the Hamilton Group was concerned, the meeting with Cassoni had no positive result. The Group agreed to meet on February 9 in Dallas. In March, the Group decided to invite IBM, a heavyweight, to join.

With Armando Stettner urging Ken Olsen, DEC hosted semi-secret meetings that included HP, IBM, Bull (France), and Nixdorf and Siemens (Germany). In May 1988, they announced the formation of the Open Software Foundation to be dedicated to the production of an operating system, a user interface, a distributed environment, and free cotton candy. Eventually, this UNIX offshoot would be AT&T license-free.

The Wall Street Journal of May 18, 1988, noted that no one at the launch of OSF could recall Ken Olsen sharing "a stage with an IBM chief executive." Ken Thompson was in Australia at the time. When Dennis Ritchie told him what had transpired, he said: "Just think, IBM and DEC in one room and we did it!" They had. But it didn't take very long for AT&T, Sun, and their coterie to form a counter-consortium, UNIX International, dedicated to the marketing of SVR4.

The war was on.

The companies that formed the OSF were joining hands to produce a new UNIX kernel and a new user interface. Their "temporary" headquarters would be in Lawrence, MA. A delegation of executives (loaned to OSF from their various corporations) attended the USENIX Conference in San Francisco in June.

OSF quickly named its executive team, including David Tory (Computer Associates) as President, and Roger Gourd (DEC), Ira Goldstein (HP), and Alex McKenzie (IBM) among the Vice Presidents.

UI appointed Peter Cunningham (ICL) as President.

By the end of 1989, Gourd's engineering team had come out with a new user interface, Motif, which was well-received, and Goldstein's research team had chosen Mach as the underlying kernel for the OS. OSF also increased its number of sponsors, adding Hitachi and Philips. However, as HP swallowed up Apollo and Siemens bought Nixdorf, at year end there were still seven sponsors.

Both OSF and UI ran membership drives and gave out pens and badges and stickers. Each ended up with about 200 members.

In 1991–92 the worldwide economy worsened. Bull, DEC, IBM, and the computer side of Siemens all lost money. AT&T resold its share of Sun. The fierce mudslinging appeared to be over. (At one point there was even a rumor of OSF and UI merging, for the good of UNIX. But that would take several more years.)

It hardly seemed to matter: Sun had adopted Motif; in 1993 USL sold UNIX to Novell, whereupon UI disbanded; OSF abandoned several of its previously announced products (shrink-wrapped software and the distributed management environment); Bull, Philips, and Siemens withdrew from sponsorship of OSF.

It was then that Armando Stettner remarked to me: "It's not clear whether there's any purpose to OSF anymore."

In 1984 a group of UNIX vendors had formed a consortium, X/Open, to sponsor standards. It was incorporated in 1987 and based in London. In 1996 OSF merged with X/Open, which owned the UNIX trademark, to become The Open Group, which then held the UNIX trademark. The Group also took on Motif and the Common Desktop Environment (CDE). The war appeared to be over.

But the Open Group maintained its concern with standards, and sponsored the Single UNIX Specification. It has also taken on sponsorship of other standards including CORBA and the Linux Standard Base.

And it was Linux that profited. At Berkeley, the CSRG began purging its distributions of copyrighted AT&T code from 1989 to 1994. Keith Bostic would convene BSD BoFs where he would announce the progress in checking that the code was free of AT&T copyright.

But BSD soon found itself in legal trouble with AT&T's USL subsidiary, who at that time were the owners of the System V code and the UNIX trademark. The *USL v. BSDi* lawsuit was filed in 1992 and led to an injunction on the distribution of BSDi's Net/2 until the validity of USL's copyright claims on the source could be judicially determined.

The lawsuit slowed development of the free-software descendants of BSD for nearly two years while the legal status was in question; as a result, systems based on the Linux kernel gained greater support.

The lawsuit was settled in January 1994, largely in Berkeley's favor. A condition of the settlement was that USL would not file further lawsuits against users and distributors of the Berkeley-owned code in the upcoming 4.4BSD release. Kirk McKusick summarized the lawsuit and its outcome:

> Code copying and theft of trade secrets was alleged. The actual infringing code was not identified for nearly two years. The lawsuit could have dragged on for much longer but for the fact that Novell bought USL from AT&T and sought a settlement. In the end, three files were removed from the 18,000 that made up the distribution, and a number of minor changes were made to other files. In addition, the University agreed to add USL copyrights to about 70 files, with the stipulation that those files continued to be freely redistributed.

Wars rarely conclude profitably for the opposing forces. Look at the lists of companies and organizations above. Apollo, DEC, Encore, Compaq, Convergent, Bull, Nixdorf, Sun, Silicon Graphics, OSF, UNIX International, etc., etc. All gone. HP, IBM, and Microsoft remain. Among the operating systems, all, even Microsoft Windows, owe debts to Bell Labs, Berkeley, and Linux.

Invisible Intruders: Rootkits in Practice

DAVID BRUMLEY

Davd Brumley then...

David Brumley works for the Stanford University Network Security Team (SUNSeT). He graduated with honors from the University of Northern Colorado in mathematics with additional work in philosophy. In his free time he enjoys playing hockey and reading Kant.

...and now

David Brumley is an Associate Professor at Carnegie Mellon University in the Electrical and Computer Engineering Department. Prof. Brumley graduated from Carnegie Mellon University with a PhD in Computer Science in 2008. Hehas received several best paper awards, an NSF CAREER award, and the United Stated Presidential Early Career Award for Scientists and Engineers. dbrumley@cmu.edu

Reprinted from *;login:* Special Issue on Intrusion Detection, September 1999

To catch a cracker you must understand the tools and techniques he will use to try to defeat you. A system cracker's first goal is to hide his presence from you, the administrator. One of the most widely used cracker tools for doing this is the rootkit. A rootkit gets its name not because the toolbox is composed of tools to crack root, but instead because it comprises tools to keep root.

Rootkits are used by intruders to hide and secure their presence on your system. An intruder achieves complete cloaking capability by relying on an administrator to trust the output of various system programs. This assumption is more or less true – most of the time system administrators trust ps to display all processes and ls to list all files.

The cracker hides simply by modifying these programs not to display his activities: ls is altered not to display the cracker's files, and ps is modified not to display the cracker's processes. This simple method proves powerfully effective. A system administrator often has no clue that anything is amiss. Should the administrator sense that the system does not "feel" right, she'll have a hard time tracking down the exact problem.

To replace any of the programs mentioned here, the cracker must already have root access. The initial attack that leads to superuser access is often very noisy. Almost every exploit will produce a lot of network traffic and/or log activity. Once in, though, the skilled attacker has no difficulty covering tracks. The average cracker will have programs in his rootkit such as z2 and wted that remove login entries from the wtmp, utmp, and lastlog files. Other shell scripts may clean up log entries in /var/log and /var/adm. Luckily, the average cracker is sloppy. Sometimes he will forget to clean out certain programs or will simply just zero out the log file. Any time a log file has zero length it should be an immediate sign that something is amiss.

Trojans

Once the cracker cleans up the appropriate files to hide his tracks, he will want to leave a backdoor in order to avoid using his noisy exploit again. Rootkit backdoors – often called trojan horses – can typically be divided into two categories: local programs and network services. These trojaned programs are the core of the rootkit.

Local programs that are trojaned often include chfn, chsh, login, and passwd. In each case, if the magic rootkit password is entered in the appropriate place, a root shell is spawned. Of course a smart cracker will also disable the history mechanism in the root shell.

The replacement for login is especially interesting. Since some systems have shadowed and unshadowed password schemes, the cracker's replacement must be of the right type. A careless cracker might use the wrong kind of login trojan. When this happens, all or some accounts will be inaccessible, which should be an immediate tipoff that a cracker has gained control of your system.

usenix 40TH ANNIVERSARY 1975–2015

inetd, the network super daemon, is also often trojaned. The daemon will listen on an unusual port (rfe, port 5002 by default in Rootkit IV for Linux). If the correct password is given after connection, a root shell is spawned and bound to the port. The manner in which the shell is bound makes it essential to end all commands with a semi-colon (";") in order to execute any command line.

rshd is similarly trojaned. A root shell is spawned when the rootkit password is given as the username (i.e., `rsh [hostname] -l [rootkit password]` will get you in to the compromised machine).

Last, a root shell is often simply left bound to a port by the program bindshell. This program requires no password. By default the program is bound to port 31337, "eleet" in cracker jargon.

Satori

In all of these programs, the default password for the newest Linux rootkit (Rootkit IV) is `satori`. Older rootkits have used `lrkr0x` and `h0tb0x` as passwords. Rarely is the default left unchanged, but it never hurts to check.

To expand their domain, the cracker may also install an Ethernet sniffer. An Ethernet sniffer listens in on all traffic on the local network, grabbing passwords and usernames destined for other machines. ifconfig will normally report such odd behavior by alerting the administrator with the PROMISC flag. Unfortunately, ifconfig is usually one of the programs modified.

The allure of rootkits should now be obvious. Even if the administrator patches the program that initially led to root access, the cracker merely has to telnet to the proper port to get a root shell. If this is closed, the cracker can try the backdoored login or rshd program. And even if that doesn't work, the cracker can still log in as a user (from perhaps a cracked password or his Ethernet sniffer) and used the trojaned ping, chfn, or chsh program to become the superuser once again.

Why do crackers break into systems? Sometimes you are targeted directly. The cracker wants information or access specifically available at your installation. Often, however, a cracker may simply want to break into any system in order to get on IRC, serve up WAREZ, or trade MP3s. If they do this, they might trojan crontab in order to hide jobs that rotate, modify, or check on the status of the illicit activity.

Hidden

What tools does the administrator have to find these trojan-horse programs? If a rootkit is properly installed, the administrator will not be able to tell the difference between the original and a modified program. A widely used cracker program named fix will take a snapshot of the system binary to be replaced. When the trojaned or modified binary is moved into place, the fix program mimics all three timestamps (atime, ctime, and mtime) and CRC checksum of the original program. A carefully constructed and compiled binary will also have the same length.

Without a cryptographically secure signature of every system binary, an administrator cannot trust that she has found the entire rootkit. If even one program goes undetected, the intruder might have a way back into your system. Several utilities, such as tripwire and RedHat's rpm, provide secure MD5 checksums of binaries. To be truly secure, the reports must be kept offline in some sort of secure location, lest the hacker tamper with the report. (Not so long ago a system-cracker magazine called Phrack published an article on defeating online tripwire reports.) These reports may be the only thing that saves you from a complete reinstallation of the entire system.

Luckily, many crackers are careless, and portions of their rootkit can be detected. The trojaned files above often have configuration files that list the programs to hide and which to display. Often they forget to hide the configuration files themselves. Since /dev is the default location for many of these configuration files, looking in there for anything that is not a normal file is often a good idea. The default setup for many rootkits is to have the configuration file begin with `pty`, such as `/dev/ptys` or `/dev/pryr`.

Another trick is to look at modification times of all programs. Although a good cracker will try to cover most of the times, they often forget a few files or directories. `find / -mtime -N -print`, where N is the number of days you expect the intruder has had access to your system, should work in most cases. I've found many times the hacker has covered his tracks well in /bin and /sbin, but left the entire build directory for his rootkit in /tmp!

Inside each modified directory you should compare the output of `echo *` with `ls`. If `ls` has been trojaned and configured to hide anything, the `echo` command will show it.

Also pay close attention to the strings in the system binaries. Although /sbin/inetd may look the right size, if the string "/bin/bash" shows up in it, you should start worrying about what else has been replaced. Another trick is to look at the file type. If file /bin/inetd says that inetd is not stripped, it most certainly has been tampered with.

If you're lucky enough to have a /proc filesystem, spend some time to become acquainted with it – there is a lot of useful

information there. By walking the directory tree you can find which processes are running. After comparing the output to what ps shows, you can determine with some level of certainty whether ps has been modified. Other files in /proc may show you all active network connections, and some others may even list all open file descriptors!

The easiest way to detect crackers, however, is to have a clean set of statically linked binaries for your system. Statically linked? Sometimes a more advanced cracker will replace system libraries, so anything that dynamically uses them cannot be trusted. If possible you should have a spare set of common programs such as ps, ls, ifconfig, perhaps lsof, etc., on a secure host. When you find a compromised system, simply download the clean binaries, set your PATH environment variable to use them, and start looking for backdoors.

Various versions of rootkit are available at most cracker sites. The most accessible versions are for open-source operating systems such as Linux and FreeBSD. Also commonly reported are versions for Irix, SunOS, and Solaris. The latest rootkit, Linux Rootkit IV, is distributed by The Crackers Layer, http://www.lordsomer.com. It is definitely worth the bandwidth to download the source and see how it works.

Rootkits have become very popular tools for both experienced and novice crackers. Your first line of defense should always be protection with regular patches and administration. Equally important is the second line: a good plan in the event of a real compromise. By arming yourself ahead of time with secure checksums and clean binaries, you will be much quicker and more effective in local and sitewide incident response.

Utilities Included in Rootkit IV

Programs That Hide the Cracker's Presence

ls, find, du—will not display or count the cracker's files.

ps, top, pidof—will not display the cracker's processes.

netstat—will not display the attacker's traffic, usually used to hide daemons such as eggdrop, bindshell, or bnc.

killall—will not kill the attacker's processes.

ifconfig—will not display the PROMISC flag when sniffer is running.

crontab—will hide the cracker's crontab entry. The hidden crontab entry is in /dev by default.

tcpd—will not log connections listed in the configuration file.

syslogd - similar to tcpd.

Trojaned Programs That Have Backdoors

chfn—root shell if rootkit password is entered in as new full name.

chsh—root shell if rootkit password is entered as new shell.

passwd—root shell if rootkit password is entered as current password.

login—will allow the cracker to log in under any username with the rootkit password. If root logins are refused, user rewt will work. It also disables history logging.

Trojaned Network Daemons

inetd—root shell listening on port rfe (5002). After connection, the rootkit password must be entered in as the first line.

rshd—trojaned so that if the username is the rootkit password, a root shell is bound to the port (i.e. rsh [hostname] -l [rootkit password]).

Cracker Utilities

fix—installs a trojaned program (e.g., ls) with the same timestamp and checksum information.

linsniffer—a network sniffer for Linux.

sniffchk—checks to make sure a sniffer is still running.

wted—wtmp editor. You can modify the wtmp.

z2—erases entries from wtmp/utmp/lastlog.

bindshell—binds a root shell to a port (port 31337 by default).

Conference Reports

In this issue:

LISA14

November 9–14, 2014, Seattle, WA

Summarized by Herry Herry and Josh Simon

Invited Talks

Making "Push on Green" a Reality: Issues & Actions Involved in Maintaining a Production Service

Dan Klein, Google, Inc.

Summarized by Josh Simon (jss@clock.org)

Despite encouragement from the audience, Dan didn't give the talk as an interpretive dance. The talk basically asked (and answered) the question, "What is 'push on green,' and what do you need to do before you can get to it?"

He laid out his assumptions for this talk: you have at least one server, at least one environment, some number of dependents or dependencies, the need to make updates, and a limited tolerance for failure.

What's "push on green"? When something—such as a human, build system, or test suite—says it's okay to release something, do it. It's unfortunately complicated by reality: how do you avoid rollout pain with a new or modified service (or API or library or...)? In summary:

- **Developing**. Peer reviewed code changes; *nobody* does a check-in-and-push (with possible exceptions when Production is broken, but the code review needs to happen after the fact). Is the code readable? Well-documented? Test your code—with both expected and unexpected conditions. Does it fail gracefully? Use new libraries, modules, and APIs; don't do a "first upgrade in five years" thing.
- **Testing**. Unit tests, module tests, end-to-end tests, smoke tests and probers, and regression tests. Find a bug? Write a test to reproduce it, patch it, and rerun the test. (Example: OpenSSL has only five simple tests at a high level and hundreds of modules that aren't directly tested at all.)
- **Monitoring**. Volume (how many hits), latency, throughput (mean, minimum, maximum, standard deviation, rate of change, and so on); historical data and graphing; alerting and service level agreements (SLAs). As a side note, SLAs require service level objectives (SLOs), which require service level indicators (SLIs).
- **Updating (and rolling back)**. Should be automated and mechanical idempotent processes. This requires static builds, ideally with human-readable version numbers like *yyyymmdd_rcn*. It needs to be correlated with monitoring. You can mark a version as "live," and then push is just changing the pointer to that live version; rollback is re-marking the "old" version as live and updating the pointer ("rolling forward to a previous version"—assuming no database schema changes anyhow). You should also have canary jobs; a canary job is in the case when you have more than one machine or process. You say "some amount of traffic will hit the canary job with the new version." You need to check the canary first to see whether it crashed. If you monitor the canaries and let them have some fraction of the traffic, you can look at those graphs and check for anomalies and trending and see whether the canary works as expected. If it looks good, you can push things live. If it doesn't, only a small fraction of users are affected for a short period of time.

Your organization needs a cross-cultural mindset across all of these.

So how do you do a safe rollout? In general:

- Silence the relevant alerts in your monitoring system.
- Update the canary jobs.
- Run your smoke tests.

- Let canaries "soak," or run for a while; the code or test might require some number of iterations such as loading a disk cache.
- Push the remaining jobs.
- Run the smoke tests again.
- Unsilence alerts.

What about making the configuration changes? You can have job restarts with runtime flags, or HUPping the job to reread the config file. The latter is faster but riskier.

In about 84 weeks with this process, Google went from roughly five rollouts per week to upwards of 60 (with a peak of 75) *and* freed up an FTE engineer. Having more frequent and smaller rollouts helps developers, who don't have to wait for the "weekly build" to release their code.

Process improvements they've made include automated recurring rollouts (using a 4.5-day week, excluding weekends and Friday afternoons), inter-rollout locking (to prevent stomping on each other), schedules of rollouts (to prevent things from happening when people are sleeping), and one-button rollbacks.

Future enhancements to the process include rollback feasibility (how easy is it to roll back the release, e.g., if there are schema changes?), continuous delivery (just release it automatically if there's a change in the binary or config file checked in), rollout quotas (prevent someone from taking all the slots for a person, team, or tool), and green on green (if there's continuous delivery and something breaks, should that halt additional deployments?).

This is an evolutionary process. Things *will* go wrong—things break, and so we need to adjust attitudes. Find the reasons why something went wrong, not to assign blame but to fix the process. Let humans be smart and machines be repeatable. Have a Big Red Button so a human can stop things if needed.

You don't need to be at Google-scale to do this. And, sorry, there are no silver bullets. It's a laborious process with lots of baby steps. You have to be careful and not take shortcuts, just keep going.

While waiting for the Q&A to start, Dan did perform an interpretive dance after all.

For more information, please see the "Push on Green" article in the October 2014 (vol. 39 no. 5) issue of *;login:*.

Distributing Software in a Massively Parallel Environment

Dinah McNutt, Google, Inc.
Summarized by Josh Simon (jss@clock.org)

Dinah McNutt has been coming to LISA since LISA IV (1990) and chaired LISA VIII (1994), and while she used to be a sysadmin she's now a release engineer. One of her passions is packaging; she's fascinated by different package managers and she talked about Google's.

The problem is that with very large networks, it may take a long time to distribute things; there are bottlenecks (such as network, disk, CPU, and memory), a machine may be offline, networks might be partitioned ("you can't get there from here"), and there are even concurrent writers.

Google's package management system is called Midas Package Manager (MPM). The package metadata (more below) is stored in their Bigtable Database, and package data is stored in their Colossus File System and replicated. The transport mechanism is a custom P2P mechanism based on torrent technology.

An MPM package and metadata contain the contents of the package (the files), a secure hash of the unique version ID, signatures for verification and auditing, labels (such as "canary," "live," "rc" with date, and "production" with date; for more on the "canary" and "live" labels see the preceding talk), pre-packaging commands, and optionally any pre- and post-installation commands.

She gave a quick case study: a config file needs to go to thousands of machines, so the relevant pieces are packaged into an MPM file, and a job (that is, a process running in a container) on each remote machine fetches and installs a new version of that MPM every 10 minutes, so the config changes can go out quickly. A post-fetch script is in the MPM to install the new config file. Easy, right?

Alas, it's not quite that simple: machines may be offline, bottlenecks must be minimized, jobs have to specify the version of a package, jobs on the same machine may use different versions of the same package, the system must be able to guarantee files aren't tampered with in flight, and the system must be able to roll back to a previous version.

At package creation, the build system creates a package definition file, which includes the file list; ownership and permissions; pre- and post-install and remove commands; and all is generated automatically. Then it runs the build command. It can apply labels and signatures at any point during the process.

If files going into the package aren't changed, a new package isn't created; the label or signature is just applied to the existing (unchanged) package.

The metadata can be both immutable (who, when, and how it was built; list of files, attributes, and checksums; some labels, especially those with equals signs; and version ID) and mutable (labels without equals signs, and cleanup policy).

The durability of an MPM package depends on its use case: test packages are kept for only three days, ephemeral packages are kept for a week (usually for frequently pushed configuration files), and durable packages are kept for three months after their last use (and stored only on tape thereafter).

Distribution is via pull (by the client-side job), which avoids network congestion (things are only fetched when needed) and

lets the job owners decide when to accept new versions (e.g., "wait until idle"). But since job owners can decide when to accept new versions, there either has to be extra logic in the job to check for new versions *or* the ability to restart jobs easily, and it can be difficult to tell who's going to be using a specific version.

The package metadata is pushed to Bigtable (which is replicated) immediately. Root servers read and cache data from their local Bigtable replica. MPM queries the local root server; failover logic is in the client, so if requests fail they're automatically redirected to another Bigtable replica.

Package data is in the Colossus File System, scattered geographically. It's a two-tiered architecture; frequently used packages are cached "nearby" (closer to the job). The fetch is via a torrent-like protocol, and the data is stored locally; so as long as it's in use you don't need to talk to either Bigtable or Colossus. There's only one copy on the machine no matter how many jobs on the machine use it. They have millions of fetches and petabytes of data moving daily.

Security is controlled via ACLs. Package namespace is hierarchical, like storage/client, storage/client/config, and storage/server. ACLs are inherited (or not). There are three levels of access:

- **Owner** can create and delete packages, modify labels, and manage ACLs.
- **Builder** can create packages and add/modify labels.
- **Label** can control who can add/modify specific labels: production.*, canary, my_label=blah, and so on.

Individual files can be encrypted within a package, and ACLs define who can decrypt the files (MPM can't). Encryption and decryption are performed locally and automatically, which allows for passwords that aren't ever stored unencrypted.

Signatures can be signed at build time or later. Secure key escrow uses the package name and metadata so a package can be verified using the name and signer.

Why love MPM? There's an mpmdiff that can compare any two packages regardless of name (like the file owner, file mode, file size, file checksums, and the pre- and post-scripts).

Labels are great. You can fetch packages using labels. You can use them to indicate where the package is in the release process (dev, canary, or production). You can promote a package by moving labels from one package to another, although some labels (those with equals signs) are immutable and can't be moved. Some labels are special ("latest," which shouldn't be used because that bypasses using a canary). They can assist in rollbacks (like "last_known_good" or "rollback" to label the current MPM while promoting the new one).

There's a concept of file groups: it's a grouping of binaries within an MPM. Binaries can belong to more than one group. Common practice is to store both stripped and unstripped binaries in the same MPM but in different file groups, to ensure the unstripped and stripped binaries match when troubleshooting problems.

There's a Web interface to browse all MPMs and show the metadata. It also shows graphs by size (so you can see how file groups change over time).

In the Q&A, Dinah addressed various questions.

Because job owners have control over when they accept new versions, the MPM team can't guarantee that every machine in production runs the "correct" version; you may have to nag people to death to upgrade. The release processes can therefore vary wildly. The SREs are good and well-respected; they're gatekeepers to keep the release processes sane. The automated build system (which is optional) enforces workflows. There is a continuous testing system where every command line submitted triggers a test. They insist that formal releases also run tests since the flags are different.

One thing possibly missing is dependency management, but that's because packages are self-contained. Performing a fetch pulls in the dependent packages, and the code explicitly lists the dependencies. In MPM, the goal was to avoid dependency management since anything can be in a package.

Building a One-Time-Password Token Authentication Infrastructure

Jonathan Hanks, LIGO Lab/California Institute of Technology; Abe Singer, Laser Interferometer Gravitational Wave Observatory, Caltech
Summarized by Herry Herry (h.herry@sms.ed.ac.uk)

Jonathan Hanks began by saying that his team built a one-time-password token authentication infrastructure, called LIGO, in part because they want to prevent credential theft before it is too late. His organization produced several requirements of the token-based solution: one token to rule them all; the token must be produced by a physical device; trust no one; and the system must be distributed, fault-tolerant, use open standards, and be cheap.

After considering the requirements, his team decided to build a custom semi-distributed system where some sites run authentication servers while others do not. The authentication system can split and join depending on circumstances. The system is using MIT Kerberos with a custom user database. The server periodically synchronizes its data with others to replicate the user database.

All services are connected to the authentication system using PAM. Each user of a service must employ a YubiKey device to generate a one-time password during authentication. The device itself is sent simply by mail, and the user must activate the device before it can be used.

Jonathan closed his talk by saying that it is important to own the system by themselves because there is no secret in the system. Thus, they can fully trust it.

JEA—A PowerShell Toolkit to Secure a Post-Snowden World

Jeffrey P. Snover, Microsoft

Summarized by Herry Herry (h.herry@sms.ed.ac.uk)

Jeffrey Snover began by illustrating that Edward Snowden is more powerful than General Michael Hayden (former CIA director), mainly because he held "the key to the kingdom": there was no limitation on what he could do on the system as a system administrator. This issue motivated Jeffrey's team to build JEA (Just Enough Admin). But although we should manage the risk, we cannot eliminate it.

Snover explained that with JEA, we can prescribe which actions that can be executed by particular admins. With this, people do not need admin privileges to do their job. He also added that all admin actions got logged, which is very useful for auditing. One of the attendees described JEA with the precise term "firewall shell."

Jeffrey mentioned that JEA is integrated inside the PowerShell. He said that we can define "command visibility" data, which drives the parsing capability of the PowerShell. This data-structure limits the commands and the parameters that can be invoked by the user.

Advanced Topics Workshop at LISA14

November 11, 2014, Seattle, WA

Summarized by Josh Simon

Tuesday's sessions included the 20th (and final) Advanced Topics Workshop; once again, Adam Moskowitz was our host, moderator, and referee. Unlike past years, we only ran for a half day. With only two new participants (both longtime LISA attendees), Adam covered each participant's interface to the moderation software in brief one-on-one sessions over lunch. We started with our usual administrative announcements. We mostly skipped introductions. However, Adam noted that two people here were at the first ATW, and he and I were both here for the past 18 years (he as moderator and I as scribe). In representation, businesses (including consultants) outnumbered universities by about two to one (about the same as last year); over the course of the day, the room included 10 LISA program chairs (past, present, and announced future, up from five last year) and nine past or present members of the LOPSA or USENIX boards.

Our first topic, which took two-thirds of the discussion time, was on why this was the last ATW. To oversimplify:

- The workshop has met its originally stated goals of increasing the number of more senior people who attend the conference and to have a space to discuss possibly confidential issues in a non-public venue with other senior people and without interruption.
- Most of the topics we discuss are not controversial and don't lead to much discussion, spirited or otherwise. There were few if any strong opinions.
- Many of the topics were repeated year after year but nothing new was being said.

Of course, since this decision was announced without input from the participants, it generated a very spirited and passionate discussion (and at times an outright debate). That discussion wandered through what the workshop should be if it were to continue, as well as the future direction of the LISA conference itself. No definitive conclusions were reached, in large part because not all stakeholders were present or represented.

It was argued that the workshop has been successful. The founder, John Schimmel, had originally looked at the conference and identified a problem: the more senior system administrators would only come to LISA (which was then more about training junior administrators) if they were speaking or teaching, and were much less likely to come as attendees. The workshop was an attempted solution to that problem: get the more senior sysadmins present for the workshop, where they could have non-public discussions without having to step down the language for junior sysadmins to understand, and they'd be (and were) much more likely to stick around for the rest of the conference.

It was also argued that there's still value in getting together, even if just "at the bar." Many were quick to point out that it would be much more difficult to sell "I'm meeting with a couple of dozen senior sysadmins at the bar" than "... at the workshop" to their management.

Some of the other points we considered during the discussion included:

- What problems are there with the conference today, and how can we solve them? In general, we see a need and provide service to the community; someone therefore needs to present a proposal (for a workshop, talk, tutorial, or whatever is appropriate) and see whether it's accepted.
- If there were no ATW, would you still attend LISA? If not, why not? How can LISA evolve to change your mind? For many people—nearly half of those present—ATW was *the* reason they attend LISA, in part because we provide seniority without overspecialization. Also, would the conference be losing something important? (Most thought so.)
- Is the workshop name still appropriate? Some years the topics aren't necessarily advanced, but we're talking about implementing and integrating existing technologies into existing (often complex legacy) environments.
- A side discussion came up as to whether we're elitist. How many of us sent in a position paper? (Virtually all at least once; it's only required the first time. At least one participant submits a position paper every year.) We need some kind of bar to keep the limited space available for the more senior

people; any bar, even if it's "senior vs. junior," can be perceived as elitist. Perhaps owning up to the word as a function of the workshop would be a good thing. Some present parsed the message of "the workshop is perceived as elitist" as "USENIX would prefer you weren't here"; the USENIX representatives present disagreed with that parsing.

- How do we, as senior sysadmins and often longtime LISA attendees, contribute to the conference?
- Why should USENIX fund this (indeed, any) workshop? Is it an appropriate use of USENIX's time and money? By reserving the room for the workshop, which loses money, there's one less room available for tutorials, which make money. That led to a discussion about what LISA should be and what USENIX should do. It was noted that USENIX has to continue to make money, to serve the community, and to serve the conference. If it's not going to be making money but serving another purpose that serves the community, that's still valid. Nobody at USENIX wants the workshop participants to stop coming, but by pulling the plug they may have that effect. The board members present agreed that they would welcome discussions with a committee elected to consider the future of the workshop.
- We need something formal on the schedule to justify our attendance and travel to our management.
- Some of our extended discussions over the years have spawned their own workshops or conferences (notably the Configuration Management workshop held at LISA and the standalone LISA-NT conference).
- This workshop is a microcosm of what happened to the general conference: it spun off other workshops and conferences and made the general conference look less important and less relevant.
- This forum of knowledgeable and experienced people is a hard resource to replace. Is this workshop, as currently constituted, the right forum for such discussion? If not, what is?
- Is the issue with the format or the lack of new blood? We only get two to three new people requesting to join each year because the position paper scares many off. That said, many agree we need new blood. One wants everyone to encourage someone new to join us next year, doubling the size of the workshop; unfortunately, workshops need to be size-limited for things to work.
- Having the same people year after year can lead to an echo chamber. Something not impacting us may be overlooked or ignored. Some of us show up out of habit; this won't be a problem as long as it doesn't drive away people with specific topics to discuss. Perhaps a position paper should be required from every participant every year (instead of only the first year)?
- How do we bring, to the conference or to the workshop if it remains, other qualified people, those on the leading edge of technologies?
- How can we be better at mentoring leaders?

It was stressed that all interesting proposals (for papers, talks, tutorials, and workshops) are both welcome and desired. A proposal saying "After *N* years we have a new version of the ATW called something else" would be considered as long as it indicated how it would be different. The number of workshops is limited by the number of rooms available and by the number of places any one of us can be at one time. It's not just what should serve USENIX or LISA better but what would serve us (the constituents) better.

As a palate cleanser we went with a lightning round: what's your favorite tool? Answers included Aptly, C+11, CSVKit, Chef, Docker, Expensify, Go, Google Docs, Graphana, Graphite, HipChat, JCubed, JIRA, R, Review Board Sensu, Sinatra, Slack, Git and git-annex, logstash, and smartphone-based cameras.

Our next discussion was about platform administrators. With user-level networking and systems becoming one blended platform, are platform admins the new sysadmins? Is this a new tier for provisioning logical load balancers and front and back ends? The conclusion seemed to be that it's still sysadmin, just a specific focus. It's like any other new technology, and may be due to the extension of virtualization into the network world. The "are we specializing?" question comes up often (e.g., storage, network, Windows versus UNIX, and so on), and we're still sysadmins.

One participant strongly disagreed, thinking platform administration is fundamentally different in that for the first time it's now readily straightforward and easy to think of system deployment as a cheap software call or RPC. It's so lightweight in so many ways that it's fundamentally different from early virtualized environments. His business expects to routinely spin up thousands of virtual instances. How much and how fast to spin things up (and down again) is a game changer. The other part of it is that the environments they're using for this are fundamentally confused about everything of value, with APIs calling APIs. At some level this is sysadmin on a new layer, because it's a programmability block mode; much of the sysadmin stuff is hidden. What happens when you're repairing a cluster and something says you have to scale out from 200 to 1000? Either "you don't" or "you wait" might be the answer.

Another person noted that we're system administrators, not just focused on the single computer (or network or person), but on the interaction between those systems (computers, networks, people, and so on). Nothing's really changed: we still look at the pieces, the goals, and whether the product/service is being delivered as expected.

Two side discussions came out of this as well. First, with virtualization and cloud and *aaS, how many businesses still administer their IT as their core function? Second, sysadmins who won't write code (including shell scripts) will soon be out of a job, since the field is moving towards that: systems will be built by writing code. With virtualization and APIs, we suspect that most sysadmins will fall into the "services" mode, maintaining

services on perhaps-dedicated, probably virtual machines, as opposed to the folks administering the underlying hardware on which the virtualized machines run.

Our next discussion was started with the phrase, "If I had a dollar for every time someone said DevOps was the future..." It took forever for Agile to get into Gartner, but DevOps is there already and, in the speaker's opinion, has jumped the shark in less than two years. DevOps is a horribly abused term, despite being a paradigm shift. At ChefConf, the belief was that DevOps was "software engineers throwing off the yoke of the evil sysadmins who have oppressed them for so long." (That's a direct quote from their keynote speaker.) Code needs to be in the realm of infrastructure; what we did 20 years ago won't scale today. There's a huge difference between writing actual code and writing a Ruby file that consists entirely of declarations.

In another company, they have some developers who do sysadmin work as well, but not all developers there have the background, and the speaker doesn't trust them to do it: their sysadmins are developers but not all developers are sysadmins.

One participant who has been going to DevOps and infrastructure-as-code meetups for a while now says it's like SAGE-AU and Sun Users' Group repeating the same mistakes all over again.

Even now, everyone has a different definition of DevOps, though most could agree it's not a tool, position, mechanism, or process, but a culture, about having the operations folks and engineers talk to each other as the product is written as well as after operations has it in production. There's a feedback loop through the entire life cycle. But having "a DevOps team" is not true; it's about not isolating teams.

We had a brief conversation on recruiting. How do you find and entice qualified people to jump ship to a new company? They have problems finding candidates who want to come to the company. The only response was that sometimes you simply can't, and one participant noted he turned down a great job because of its location (being sufficiently unpleasant to make it a deal breaker).

We then discussed what tools people are using to implement things within a cloud infrastructure. One participant is all in AWS, for example. Do you do it manually or through automation, what do you use to track things and manage things, and so on? One participant snarked he'd have an answer next year.

Another is about to start moving away from the AWS API to the Terraform library (written in Go), which supports several different cloud vendors and has a modular plugin system. Beyond that it depends on what you're trying to do.

Yet another says part of this is unanswerable because it depends on the specific environment. His environment is in the middle of trying to deploy OpenStack storage stuff, and most of the tools can't work because they reflect the architectural confusion thereof. They have used ZeroMQ for monitoring and control due to scalability to a million servers—which is what they call a medium-sized application. Precious few libraries can handle that level. That's the number thrown around by HPC too.

Once you care about speed and latency and measurements you can make a better judgment of how much to spin up to handle those requirements and whether physical or virtual is the right answer for your environment.

Our final discussion topic was on getting useful information from monitoring data.

One participant loves Graphite. Since he has a new hammer everything looks like a thumb, so he's been trying to get more and more into it, and now that he's taken the stats classes, he needs more low-level information so he can draw correlations and eventually move data out of the system. What are others doing with their statistics? What are you using to gather, store, and analyze data? In general, R and Hadoop are good places to start, and there's an open source project called Imhotep for large-scale analytics. Several others noted they use Graphite as a front end to look at the data. Spark is useful for real time and streaming. Nanocubes can do real-time manipulation of the visualization of a billion-point data set. Messaging buses discussed included RabbitMQ and ZeroMQ.

How does this help? In one environment, they used the collected metrics to move a datacenter from Seattle to San Jose, and the 95th percentile improved a lot. Another noted that Apple determined that the transceiver brand makes a huge difference in performance.

We wrapped up with the traditional lightning round asking what we'd be doing in the next year. Answers included an HPC system with 750K cores and an 80 PB file system, automation and instrumentation, chainsaws and hunting rifles in Alaska, enabling one's staff, encouraging people to create and follow processes, exabyte storage, functional programming, Hadoop, home automation, Impala, infrastructure, learning a musical instrument, merging an HPC-focused staff into the common IT group, moving from GPFS to something bigger, network neutrality, organizing a street festival and writing the mobile app for it, packaging and automated builds, producing a common environment across any type of endpoint device, R, scaling product and infrastructure (quadrupling staff), Spark, trying to get the company to focus on managing problems not incidents, and updating the Cloud Operational Maturity Assessment.

Our moderator thanked the participants, past and present, for being the longest-running beta test group for the moderation software. The participants thanked Adam for moderating ATW for the past 18 years.

29th Large Installation System Administration Conference (LISA15)

Sponsored by USENIX, the Advanced Computing Systems Association, in cooperation with LOPSA

November 8–13, 2015, Washington, D.C.

Important Dates

- Submissions due: April 17, 2015, 11:59 pm PDT
- Author response period: May 11, 2015–May 17, 2015
- Notification to authors: June 5, 2015
- Final papers and poster abstracts due: August 27, 2015, 11:59 pm PDT

Conference Organizers

Program Co-Chairs

Cory Lueninghoener, Los Alamos National Laboratory

Amy Rich, Mozilla Corporation

Overview

Publishing at USENIX LISA Means Industry Impact and Recognition

Papers published at LISA receive visibility and recognition from a unique audience that fundamentally builds and operates the systems that run the world. For a researcher, publishing at LISA proves that your work is relevant and applicable to industry and has the potential to transform the profession's approach to systems engineering.

USENIX Stands for Open Access and Advancing the State of the Computing World

USENIX strongly believes that research is meant for the community as a whole and maintains a clear stance advocating open access. Your poster abstract or paper and presentation will be available online at no charge, where it can have the most impact and reach the broadest possible audience. Papers and poster abstracts will also be part of the conference proceedings, which has its own ISBN.

Topics of Interest

- Systems and Network Engineering
  - Cloud and hybrid cloud computing
  - Software Defined Networks (SDN)
  - Virtualization
  - HA and HPC clustering
  - Cost effective, scalable storage
  - Configuration management
  - Security
- Monitoring and Metrics
  - Performance and scalability
  - Monitoring, alerting, and logging systems
  - Analytics, interpretation, and application of system data
  - Visualization of system data
- SRE/Software Engineering
  - Software-defined System Administration
  - Continuous delivery and product management
  - Continuous deployment and fault resilience
  - Release engineering
  - Big Data
- Culture
  - Business communication and planning
  - Standards and regulatory compliance
  - DevOps
  - On-call challenges
  - Distributed and remote worker challenges
  - Mentorship, education, and training
  - Workplace diversity

Papers

Refereed papers accepted to LISA describe new techniques, tools, theories, and inventions, and present case histories that extend our understanding of system and network administration. They present new ideas, backed by rigorous and repeatable methodology, in the context of previous related work, and can have a broad impact on operations and future research.

Accepted papers will be published in the LISA15 proceedings and conference web pages, and all accepted papers will be given a presentation and Q&A slot in the LISA15 academic research track. After acceptance, paper authors should also create a companion poster to present during the poster session to further engage with the conference attendees. Cash prizes will be awarded at the conference for the best refereed paper.

Posters

The LISA juried posters session is a great opportunity to present your work visually, either as a companion to a refereed paper or as a standalone submission displaying work in progress. Posters present the core of your research to a wide audience, providing you with the ability to gain valuable feedback from both industry and academic audiences. Poster abstracts of 500 words or less will be published in the LISA15 Proceedings, and a link to your research website will be included in the conference web page. Additionally, cash prizes will be awarded at the conference for the best juried poster.

We hope that you will consider sharing your ongoing research with the LISA community. All paper and poster abstract submissions are due by April 17, 2015. If you have any questions beyond this CFP, please contact lisa15research@usenix.org.

Paper and Poster Submission Rules

Submissions may only be submitted by an author, no third parties. Authors retain ownership of the copyright to their works as described in the USENIX Conference Submissions Policy at www.usenix.org/usenix-conference-submissions-policy.

Submissions whose purpose is to promote commercial products or services will be rejected.

Papers and poster abstracts may not be simultaneously submitted to other venues. Writing must be original and not previously published online or otherwise. If the content is similar to, or builds upon prior work, the author must cite that work and describe the differences.

The first page of each paper submission or poster abstract submission must include the name, affiliation, and email address of the author(s).

All paper submissions must be full papers, in draft form, 8–12 pages in length, including diagrams, figures, full references, and appendices. All poster abstracts must be 500 words or less.

All draft papers and poster abstracts must be submitted as PDFs via the Web form, which is located at https://lisa15.usenix.hotcrp.com/. They must be in two-column format, using 10-point type on 12-point (single-spaced) leading, with a maximum text block of 6.5" wide x 9" deep, with .25" inter-column space, formatted for 8.5" x 11" paper. Pages must be numbered, and figures and tables must be legible when printed. Templates are available for Microsoft Word and LaTeX at www.usenix.org/templates-conference-papers. Papers not meeting these criteria will be rejected without review.

Paper and Poster Acceptance Details

All submitters will be notified of acceptance or rejection by June 5, 2015.

Authors of accepted papers will be assigned one or more shepherds who will read and offer advice on intermediate drafts before submission of the final paper. At least one author must present at the LISA conference, and the chosen presenter(s) must rehearse their presentation with their shepherd prior to the conference.

One author per accepted paper will receive a registration discount.

Final poster abstracts and papers must be submitted for publication by August 27, 2015, and completed posters are required by the start of the conference. Accepted presenters should ensure that they have enough time to acquire the necessary approvals through their organizations' Institutional Review Board (IRB) or similar process in time for the final paper submission deadline.

If any accepted presenters need an invitation letter to apply for a visa to attend the conference, please identify yourself as a presenter and include your mailing address in an email to conference@usenix.org as soon as possible. Visa applications can take at least 30 working days to process.

All accepted papers and poster abstracts will be available online to registered attendees before the conference. If your work should not be published prior to the event, please notify production@usenix.org when you submit your final copy.

usenix

LISA15

Nov. 8 – 13, 2015 | Washington, D.C.

Sponsored by USENIX in cooperation with LOPSA

SRE

CULTURE

METRICS

Industry Call for Participation

LISA is the premier conference for IT operations, where systems engineers, operations professionals, and academic researchers share real-world knowledge about designing, building, and maintaining the critical systems of our interconnected world.

We invite industry leaders to propose topics that demonstrate the present and future of IT operations. LISA submissions should inspire and motivate attendees to take actions that will positively impact their business operations.

Important Dates

Submissions due: **April 17, 2015, 11:59 pm PDT**
Notification to participants: **June 5, 2015**

Topic Categories

Systems and Network Engineering

- Cloud and hybrid cloud computing
- Software Defined Networks (SDN)
- Virtualization
- HA and HPC clustering
- Cost effective, scalable storage
- Configuration management
- Security

Monitoring and Metrics

- Performance and scalability
- Monitoring, alerting, and logging systems
- Analytics, interpretation, and application of system data
- Visualization of system data

Conference Organizers

Program Co-Chairs

Cory Lueninghoener, Los Alamos National Laboratory
Amy Rich, Mozilla Corporation

SRE/Software Engineering

- Software-defined System Administration
- Continuous delivery and product management
- Continuous deployment and fault resilience
- Release engineering
- Big Data

Culture

- Business communication and planning
- Standards and regulatory compliance
- DevOps
- On-call challenges
- Distributed and remote worker challenges
- Mentorship, education, and training
- Workplace diversity

Proposals we are Seeking

- **Talks:** 30-minute talks with an additional 10 minutes for Q&A
- **Mini-tutorials:** 90-minute courses teaching practical, immediately applicable skills
- **Tutorials:** Half-day or full-day courses taught by experts in the specific topic, preferably with interactive components such as in-class exercises, breakout sessions, or use of the LISA lab space
- **Panels:** Moderator-led groups of 3–5 experts answering moderator and audience questions on a particular topic
- **Workshops:** Half-day or full-day organizer-guided, round-table discussions which bring members of a community together to talk about common interests
- **Papers and posters:** Showcase for the latest research; see the Academic Research Papers and Posters Call for Participation at www.usenix.org/lisa15/cfp/papers-posters for more information

Submit your proposal today!
www.usenix.org/conference/lisa15/call-for-participation/submission

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 format, in Mobi format, on-line and through our Android and iOS apps. Wherever you go, *Linux Journal* goes with you.

SUBSCRIBE NOW AT:

www.LinuxJournal.com/subscribe

usenix
THE ADVANCED
COMPUTING SYSTEMS
ASSOCIATION

USENIX Association
2560 Ninth Street, Suite 215
Berkeley, CA 94710

POSTMASTER
Send Address Changes to *;login:*
2560 Ninth Street, Suite 215
Berkeley, CA 94710

PERIODICALS POSTAGE
PAID
AT BERKELEY, CALIFORNIA
AND ADDITIONAL OFFICES

usenix 40TH ANNIVERSARY

usenix
THE ADVANCED
COMPUTING SYSTEMS
ASSOCIATION

Register Today!

2015 USENIX Annual Technical Conference

JULY 8–10, 2015 • SANTA CLARA, CA
www.usenix.org/atc15

USENIX ATC '15 brings leading systems researchers together for cutting-edge systems research and unlimited opportunities to gain insight into a variety of must-know topics, including virtualization, system administration, cloud computing, security, and networking.

Co-located with USENIX ATC '15: July 6–7, 2015

HotCloud '15

7th USENIX Workshop on Hot Topics in Cloud Computing
www.usenix.org/hotcloud15
Researchers and practitioners at HotCloud '15 share their perspectives, report on recent developments, discuss research in progress, and identify new/emerging "hot" trends in cloud computing technologies.

HotStorage '15

7th USENIX Workshop on Hot Topics in Storage and File Systems
www.usenix.org/hotstorage15
HotStorage '15 is an ideal forum for leading storage systems researchers to exchange ideas and discuss the design, implementation, management, and evaluation of these systems.