;login:

APRIL 2014 VOL. 39, NO. 2

Sysadmin and Networking

usenix

THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

UPCOMING EVENTS

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

HotCloud '14: 6th USENIX Workshop on Hot Topics in Cloud Computing
June 17–18, 2014
www.usenix.org/hotcloud14

HotStorage '14: 6th USENIX Workshop on Hot Topics in Storage and File Systems
June 17–18, 2014
www.usenix.org/hotstorage14

9th International Workshop on Feedback Computing
June 17, 2014
www.usenix.org/feedbackcomputing14

WiAC '14: 2014 USENIX Women in Advanced Computing Summit
June 18, 2014
www.usenix.org/wiac14

ICAC '14: 11th International Conference on Autonomic Computing
June 18–20, 2014
www.usenix.org/icac14

USENIX ATC '14: 2014 USENIX Annual Technical Conference
June 19–20, 2014
www.usenix.org/atc14

UCMS '14: 2014 USENIX Configuration Management Summit
June 19, 2014
www.usenix.org/ucms14

URES '14: 2014 USENIX Release Engineering Summit
June 20, 2014
www.usenix.org/ures14

23rd USENIX Security Symposium
August 20–22, 2014, San Diego, CA, USA
www.usenix.org/sec14

Workshops Co-located with USENIX Security '14

EVT/WOTE '14: 2014 Electronic Voting Technology Workshop/Workshop on Trustworthy Elections
August 18–19, 2014
www.usenix.org/evtwote14

USENIX Journal of Election Technology and Systems (JETS)
Published in conjunction with EVT/WOTE
www.usenix.org/jets

CSET '14: 7th Workshop on Cyber Security Experimentation and Test
August 18, 2014
www.usenix.org/cset14
Submissions due: April 25, 2014

3GSE '14: 2014 USENIX Summit on Gaming, Games, and Gamification in Security Education
August 18, 2014
www.usenix.org/3gse14
Invited submissions due: May 6, 2014

FOCI '14: 4th USENIX Workshop on Free and Open Communications on the Internet
August 18, 2014
www.usenix.org/foci14
Submissions due: May 13, 2014

HotSec '14: 2014 USENIX Summit on Hot Topics in Security
August 19, 2014
www.usenix.org/hotsec14

HealthTech '14: 2014 USENIX Summit on Health Information Technologies
Safety, Security, Privacy, and Interoperability of Health Information Technologies
August 19, 2014
www.usenix.org/healthtech14
Submissions due: May 9, 2014

WOOT '14: 8th USENIX Workshop on Offensive Technologies
August 19, 2014
www.usenix.org/woot14

OSDI '14: 11th USENIX Symposium on Operating Systems Design and Implementation
October 6–8, 2014, Broomfield, CO, USA
www.usenix.org/osdi14
Abstract registration due: April 24, 2014

Co-located with OSDI '14:

Diversity '14: 2014 Workshop on Supporting Diversity in Systems Research
October 5, 2014

HotDep '14: 10th Workshop on Hot Topics in Dependable Systems
October 5, 2014

INFLOW '14: 2nd Workshop on Interactions of NVM/Flash with Operating Systems and Workloads
October 5, 2014

TRIOS '14: 2014 Conference on Timely Results in Operating Systems
October 5, 2014

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 2014 VOL. 39, NO. 2

usenix
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

EDITOR
Rik Farrow
rik@usenix.org

COPY EDITORS
Steve Gilmartin
Amber Ankerholz

PRODUCTION MANAGER
Michele Nelson

PRODUCTION
Arnold Gatilao
Casey Henderson

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.

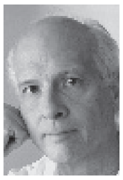
$90 of each member's annual dues is for a subscription to *;login:*. Subscriptions for non-members are $90 per year. Periodicals postage paid at Berkeley, CA, and additional offices.

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

EDITORIAL

Musings

RIK FARROW



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

I have a new hat, one with a fancy feather in it, perhaps to help it feel lighter. I've become the tutorial manager, taking over for Dan Klein after he's done the job single-handedly for more than 20 years. I appreciate what Dan has done and will have to work hard, working with a team, to do as well and, hopefully, better. This change has led me on a quest for information about the future of system administration.

What I learned is nothing new, in a way: system administration has always been changing. But, as always, there will be the risk takers, the early adopters, and the adventurers—the ones who strike out in new directions expecting that others will surely follow them.

There are also those who, having been through several schools of hard knocks, would like to continue on the path that they have become adept at navigating. Those hard knocks were painful and time-consuming, and learning new ways of doing things means going through yet another learning process.

Being an older kind of guy, I too can resist change. Then again, I know it's good for the ol' noggin to stretch its limits by learning new things. I've watched what being really stuck does to older folks than I, and I surely don't want to go there.

System Administration, Then and Now

Like Elizabeth Zwicky writes in this issue, I am not a system administrator. Oh, I can still fake it, by managing my own systems (DNS, SMTP, DHCP, and HTTP) and occasionally even consulting locally. But I gave up on being a sysadmin when I found out that I wasn't very interested in a very important aspect of it: building scripts and systems to automate the work that needed to be done routinely.

What I liked most about being a sysadmin was solving problems. Although my skills are dwarfed by Brendan Gregg (also in this issue), I followed one of the techniques in his book [1], the scientific method. I didn't learn it from his book, but from Arthur Conan Doyle's fictional detective, Sherlock Holmes. I read all the stories about Sherlock, and what stuck with me was a methodical approach to problem solving: gathering evidence, creating a hypothesis, testing it, then either fixing the problem or iterating, depending on the result.

I think I also liked the effect I could have on people when I produced wizard-like successes. Often, simply by creating a clear problem statement, the answer to the problem would just pop out. I liked feeling and appearing smart.

What I didn't like was the boredom of repetition: adding a user to a handful of workstations or performing updgrades on that same set of systems. Like Elizabeth Zwicky, I am not a programmer who builds systems from scratch. So I contented myself with fixing systems, sometimes by building scripts for specific purposes. Managing a group of systems through a unifying set of programs, however, was beyond my thinking. I gradually drifted away from system administration.

For most of the '90s, system administration remained very similar to what I had experienced. People managed each system separately, perhaps leveraging some network software (like NIS or LDAP) to solve part of the management issues. Someone who managed a lot of systems, say one or two hundred, would do so by making them all look exactly alike, whether they were part of a cluster or developer desktops.

Configuration management brought about some real changes, allowing administration of more systems without the need to make them all the same: now we could have groups of systems, and systems with sets of software on them. When I found myself installing 30 laptops for classes I was teaching, I couldn't have done it without CFEngine. And I used CFEngine as an example of how to properly manage networks of systems in those very classes.

Scale

I've always been fascinated by big machines, whether they were harvesters, bulldozers, or computers that filled a large room. I got to visit MCI's NOC, via connections to UUNet, and could marvel at the volume of data moving through a subterranean room in Northern Virginia.

But there was something else beginning to happen in the early noughts, the birth of services run on immense farms of distributed systems. Amazon and Google were experiencing tremendous growth, and dealing with this by building networks of computers on a scale never before seen or heard of. These networks might have been managed like clusters and supercomputers had been. But they weren't. These systems weren't there for running batch jobs, but for providing services to users who expected response times in terms of seconds, or less. The '90s clusters weren't designed for this, and although the tiered systems—load balancers in front of Web servers in front of a database—provided a rough model, they would freeze under the loads that these new companies were experiencing.

Todd Underwood, in his closing plenary at LISA '13 [2] and in an article in this issue, addresses the human side of scale. Quite simply, you can't, and don't want to, scale human sysadmins at the same rate you scale servers. Doug Hughes explains (in this issue) how D. E. Shaw Research buys racks of configured servers at a time, and that they figured out how they could autoconfigure both the racktop switch and the servers as soon as they were plugged in. In my interview with John Looney, you get an even better notion of how scale affects how work gets done. Instead of spending months setting up a new datacenter, a Google team spent months building automation. That automation does in just three hours what took teams of people months to complete.

I don't believe that system administration, as it has been practiced, is going to disappear. But I certainly know that system administrators are going to need new skills so they can work with online services. Debugging a distributed system where you have access to both the networks and the servers is very different from troubleshooting a distributed application running in the cloud where your access is limited. And working at scale, even modest scales, means the end of manually searching logs and monitoring: you need automation to help not just pick up problems but respond to them appropriately, and in a timely manner.

Will every service move to a cloud? I don't think the NSA is going to, and neither will privacy-sensitive or regulatory-bound businesses. People will still be running company networks and servers, and in some cases, like D. E. Shaw Research, their own supercomputers. Cloud computing works very well for many things, especially Internet services. But office, factory, and many other workers will continue to rely on internally provided services—ones that will keep working even when their connection to the Internet has failed or a cloud service becomes temporarily unavailable.

And we will not all be working at Google or some place like it: not hardly, as working at these scales does take some skills that not everybody has or wants to spend time doing. But I know one thing for certain: system administration will continue to evolve.

The Lineup

We start off this issue with a deliberately provocative piece by Todd Underwood. Todd emphatically states that the practice of system administration needs to change, which, whether you agree with Todd or not, is definitely happening. What isn't as clear is just how system administration will evolve, or how much it will follow Google's lead.

Next up is an interview with John Looney. John is an SRE for Google and taught the SRE Classroom class at LISA '13. I missed visiting this class, because I was teaching the same day, and I learned about it from hearing people talk about it over the next couple of days.

John provides the clearest explanation, through examples, of how being an SRE is different from what sysadmins, even those administering to large clusters, do. His description of how different things are within Google is crystal clear for me, and I think it will be useful to you as well.

Elizabeth Zwicky volunteers her own view: although Elizabeth started out as a system administrator, that was a doorway for her into many different jobs over the years, some related to system administration, and some barely at all.

Dinah McNutt, past LISA chair, writes about release engineering, a field related to system administration. According to Dinah, release engineering is critical for any company providing a service in a competitive market, and there are many subfields related to release engineering as well.

I interviewed Tom Hatch, creator and principal of SaltStack. SaltStack can be thought of as a grid execution engine, but it can do much more. I wanted to ask Tom about his use of ZeroMQ for message queueing and learn more about where he plans to take SaltStack.

Brendan Gregg provides this issue's troubleshooting feature. Brendan takes us through debugging a performance issue, explaining along the way how he performs troubleshooting. You can consider this a taste of what you'll find in his book [1].

David Lang continues his series on logging, focusing this time on Splunk. Splunk provides some wonderful features, but it does need to be set up and tuned if it is to provide the best performance. And, setting your Splunk servers up incorrectly can fool you into thinking you need more servers, when you simply need to do things differently.

Andy Seely begins a series of articles about managing system administrators and other IT staff. Andy explains how he handled a situation where his management needed someone to blame for a failure that was really outside the scope of what his own organization had control over.

Changing focus, Rob Sherwood updates us on Software Defined Networks (SDNs). Rob wrote for *;login:* in February 2011 [3] about safely using SDN in a production network. For this issue, I asked Rob to provide us with an update on where SDN is today, and why it is important to know about, even for those with moderately sized networks.

Doug Hughes and his co-workers have taken a careful look at the most commonly used DHCP daemon and found it lacking. They needed the DHCP protocol to do something they thought it could easily do, and they found that the easiest way forward was to do it themselves. Along the way, Doug explains DHCP, what it can do, and what they needed it to do to support their organization.

David Blank-Edelman stays on the theme of Perl and NoSQL databases. In this column, he explains MongoDB and explores its Perl interface. MongoDB is different in many ways from Redis, which he explored in the previous issue.

David Beazley discusses Python 3 from the perspective of whether you should now be using it or porting existing Python 2 code to Python 3. David provides a very balanced answer, as well as reminders for tools and techniques for moving to Python 3.

Dave Josephsen takes us on a graphic journal through a tool that helps in collaborative troubleshooting. Dave shows examples of ChatOps, a chat tool that includes the ability to include graphs and links, useful in operations.

Dan Geer and Richard Bejtlich explore a method for understanding the risk of data theft: counting and classifying digital security incidents, and measuring the time elapsed from the moment of detection to the moment of risk reduction. As usual, Dan and his co-author take a hard-line approach that is also much more realistic than common practice.

Robert Ferrell also writes about being a system administrator this time. He muses about his past as a sysadmin and pines for the power he has given up.

My usual book reviewers surprised me this month with just three book reviews. I've added one of my own, on a book that I really liked.

System administration isn't dead or dying. It's changing. Just like it always has, as technology has advanced. To be honest, you really do not want to do what I did in the mid-'80s: wire up a hub-and-spoke arrangement of serial connections to support UUCP. UUCP did relay email and support file transfers, but the truth was that it was all we had. Within a handful of years, TCP/IP networks became the new norm, and within 15 years, people would be discussing sharing DRAM over the network instead of using local disks.

In 1988, AT&T was releasing System V Release 4 (SVR4), with a focus on GUI interfaces for system administration. Under the hood, some of the commands for managing configuration had gotten extremely ugly (multiple lines with many strange options) to do what had previously been simple. The designers' argument was that system administrators shouldn't be hand editing these files, as it wasn't safe. I replied that hand editing files may not be safe, but GUIs and impenetrable syntax both restrict flexibility and cannot work with scripting.

Bay Networks was a competitor to Cisco in the early '90s, and they too had a very nice GUI for managing their routers. But their interface was also restrictive. Cisco had (and still has) their command-line management for network gear, which was tremendously more flexible than Bay Networks. Today, Bay Networks is just a memory.

SDN promises a much more flexible approach to network management, and I expect it will have a similar effect on old school network equipment companies.

In the world of sysadmin, we now have tools like Augeas [4] that turn the myriad formats of configuration files into tree-like structures, helping to hide their eccentricities.

I had considered updating the story of Chuck, my system administrator of the near future [5]. But I kept seeing Chuck getting bored and fat (from all the bon-bons he was eating) because his job had been taken over by automation.

I don't believe things are that bad. We still have time to shape our future, which we can do through the tools we build, and the philosophies behind those tools. Google employees have represented one view of the future, where automation is emperor, and I think that their view is not just relevant, but a likely future.

And, like Todd Underwood, I don't think humans will be forced out of the picture, at least, not for a long time. Not only are humans writing the automation tools, they are also troubleshooting what goes wrong with much more flexibility than exists in any automation tool.

References
[1] Brendan Gregg, *Systems Performance for Enterprise and Cloud* (Prentice Hall, 2013).

[2] Todd Underwood, "PostOps: A Non-Surgical Tale of Software, Fragility, and Reliability," 27th Large Installation System Administration Conference (LISA '13): https://www.usenix.org/conference/lisa13/technical-sessions/plenary/underwood.

[3] Rob Sherwood, "Safely Using Your Production Network as a Testbed," *;login:*, February 2011, vol. 36, no. 1: https://www.usenix.org/publications/ login/february-2011-volume-36-number-1/safely-using-your-production-network-testbed.

[4] Augeas, a configuration API: http://augeas.net/.

[5] Musings about the future of sysadmin, featuring Chuck, *;login:*, February 2010, vol. 35, no. 1: https://www.usenix.org/publications/login/february-2010-volume-35-number-1/musings.

Letter to the Editor

Dear Editor,

I want to congratulate you for the very nice and timely focus on file systems (and object storage in particular) in the *;login:* February 2014 issue.

I appreciate the first article's ("A Saga of Smart Storage Devices") overall analysis of the landscape of Object Storage but as someone who was closely involved in the design of Ceph I feel the need to express my confusion by how Ceph was portrayed.

The article describes Ceph as "a cluster of OSDs cooperating to perform automatic replication, recovery, and snapshots" that "builds on previous work," presumably NASD, "decentralized placement of objects on devices" and "the RUSH family of algorithms."

I'd like to make two clarifications:

1. Ceph is a parallel file system and, as such, much more than a cluster of cooperating OSDs. The following papers give a more accurate account (none of them were cited in the article): [1-4]. The authors appear to conflate "Ceph" with the Ceph's object storage subsystem which is called RADOS (Reliable Autonomic Distributed Object Store, described in detail in [5], also not cited) and mention the term RADOS only in the context of a protocol.
2. Similarly, while Ceph's data placement (aka CRUSH) is in part related to RUSH, RUSH turned out to be an insufficient solution in practice. CRUSH fully generalizes the useful elements of RUSH while resolving previously unaddressed reliability and replication issues, and offering improved performance and flexibility. The authors do write about the various features of CRUSH but without mentioning it or citing the paper describing it [6].

Thanks,
Carlos Maltzahn
carlosm@soe.ucsc.edu

References
[1] S. A. Weil, K. T. Pollack, S. A. Brandt, and E. L. Miller, "Dynamic Metadata Management for Petabyte-scale File Systems," in SC '04 (Pittsburgh, PA), ACM, Nov. 2004.

[2] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn, "Ceph: A Scalable, High-performance Distributed File System," in OSDI '06 (Seattle, WA), Nov. 2006.

[3] K. V. Shvachko, "HDFS Scalability: The Limits to Growth," *;login:*, vol. 35, no. 2, 2010.

[4] C. Maltzahn, E. Molina-Estolano, A. Khurana, A. J. Nelson, S. A. Brandt, and S. A. Weil, "Ceph as a Scalable Alternative to the Hadoop Distributed File System," *;login:*, vol. 35, no. 4, 2010.

[5] S. A. Weil, A. Leung, S. A. Brandt, and C. Maltzahn, "Rados: A Fast, Scalable, and Reliable Storage Service for Petabyte-scale Storage Clusters," in Proceedings of the 2007 ACM Petascale Data Storage Workshop (PDSW '07), (Reno, NV), November 2007.

[6] S. A. Weil, S. A. Brandt, E. L. Miller, and C. Maltzahn, "CRUSH: Controlled, Scalable, Decentralized Placement of Replicated Data," in SC '06 (Tampa, FL), ACM, Nov. 2006.

OPINION

The Death of System Administration

TODD UNDERWOOD

Todd Underwood is a site reliability director at Google. Before that, he was in charge of operations, security, and peering for Renesys, a provider of Internet intelligence services; and before that he was CTO of Oso Grande, a New Mexico ISP. He has a background in systems engineering and networking. Todd has presented work related to Internet routing dynamics and relationships at NANOG, RIPE, and various peering forums. tmu@google.com

I come to bury system administration, not to praise it.
The evil that well intentioned, clever geeks do lives after them;
The good is oft interred with their bones (and their previous jobs);
So let it be with system administration. The noble devops
Hath told you sysadmins were ambitious but insufficiently so:
If it were so, it was a grievous fault;
And grievously hath system administrators answer'd it.
Here, under leave of software engineers and the rest, —
For we are an honorable people;
So are they all, all honorable sysadmins, —
Come I to speak in sysadmins's funeral.
It was my friend, faithful and just to me and to all of us...

—with apologies to Bill Shakespeare

We are in the final days of system administration. I'm as nostalgic as many of you are. But I hope it is the same nostalgia that causes some of us old folks to fetishize terrible, unsafe, inefficient cars from the 1960s—a nostalgia borne of history but with no designs on the future. I hope we can cherish our shared history and accomplishments and then put them to rest in favor of a future where computers work for us rather than us for them. I dream of a future where we stop feeding the machines with human blood.

In the Beginning...ISPs

I grew up as a sysadmin first at a university and then at an Internet Service Provider (AS2901) during the 1990s. This is just what we did at the dawn of the Internet age. ISPs were a fantastic place to learn. We were all working massively beyond our capabilities and knowledge, gleefully skating over the edge of competence. At my ISP, we ran Sendmail (didn't you?). Everyone was still configured as an open relay, of course. One of the first things I did when I started was teach a class in locking down Sendmail. I didn't know how to lock down Sendmail. Obviously, I had to learn sendmail.cf syntax, m4, and at least a little bit about what the heck these spammers were doing with our mail systems. And I needed to teach a class on these things. In three days. Those were heady days.

Everyone did some of everything. The networking staff ran their own name servers, BIND 4 on AIX and FreeBSD. The systems staff often ran our own cable, and sometimes terminated it for ourselves, badly with plenty of near-end crosstalk. And the developers (sometimes) did their own backups. We were all learning and all building something that had never been seen before. As we went, we looked for more and better solutions. By the fourth time we had to modify our "new user creation" script, we were already pretty sick of it. We wanted centralized, automatable authentication and authorization. NIS? NIS+? RADIUS + some custom DB2 module (hat tip to Monte Mitzelfeld here)? LDAP! And that was better. Not perfect, just better. We traded the challenge of maintaining buggy, custom software for the challenge of maintaining, integrating, and automating LDAP. But things were moving on up.

It didn't help very much for the Windows hosting customers, and it didn't fix the entries in the billing database perfectly. But it was better.

At my next gigs, I tried to consistently automate away the toil, but the trivial toil (editing crontab by hand) was often replaced by larger scale toil (maintaining yum repositories, updating installer infrastructure, editing and testing Puppet configs). The toil became more interesting and the scale grew considerably, but I realized that the work itself didn't seem substantially better.

Site Reliability Engineering at Google

Google chose to solve this set of problems very differently. They invented something that's rapidly becoming an industry-standard role: Site Reliability Engineering. The history of exactly how and why that happened isn't written down and I wasn't there, which leaves me perfectly free to reimagine events. This almost certainly isn't what happened, but it is completely true.

Google was, and still is, incredibly ambitious and incredibly frugal. They had dreams of scaling beyond what anyone had seen before and dreams of doing it more affordably than anyone thought possible. This meant avoiding anything with costs that scaled linearly (or super-linearly) to users/queries/usage. This approach applied to hardware, and it most certainly applied to staff. They were determined to not have a NOC, not have an "operations" group that just administered machines. So, in the early days, software developers ran their own code in production, while building software to create the production environment.

Developers didn't like that very much, so they proceeded to automate away all of the annoying drudgery. They built automated test, build, push infrastructure. They built a cluster operating system that handled task restarting, collected logs, deployed tasks to machines, and did distributed tracing/debugging. Some of the software engineers working during these early days were much more production-oriented than others. These became the first site reliability engineers.

SRE at Google is composed of systems and software engineers who solve production problems with software. That's how it started and how it still is.

SRE != DevOps

The DevOps cultural movement that has been happening over the past few years is a huge improvement over the culture of system administration, IT, and operations that came before. It emphasizes collaboration among several siloed organizations and closer interaction between developers and the production environment they will work on. The rise of DevOps coincided with the conclusion of a many-year sysadmin obsession with configuration management systems (CFEngine, Puppet, anyone?). Sysadmins finally agreed that OS configuration management was drudgery and was best dealt with using configuration management tools.

As adoption and recognition of DevOps has grown, we have seen artificial and counterproductive barriers among organizational divisions fall. Well, not "fall," but at least "become somewhat more porous." We've also seen a bevy of software developed in the spirit of DevOps and claiming the DevOps mantle. This is mostly just marketing, where "DevOps" and "Cloud" are words that don't mean much more than "a buzzword to get this PO approved."

Still, almost everything is better now. We're not romanticizing the days when grizzled sysadmins built user accounts by useless-use-of-cat-ing individual bytes into /etc/passwd and /etc/shadow files by hand. We are slowly growing to realize that our human potential is much better suited to actual creative endeavors. More building; more thinking; less futzing.

But we're not even close to done. If there's one significant failing of the DevOps movement/community, it's that they don't hate operations nearly enough. They want to make operations better and to embed operational concerns and perspectives into software development, security, and networking. I think that's a good start along the way to killing operations entirely, for software-centric, Internet-built infrastructures.

Adrian Cockcroft of Netflix coined the term "NoOps" [1] to describe this vision of an operations-free future. I referred to a similar vision as "PostOps" recently [2]. This is the ambitious future, and near-present, that I think we should be collectively demanding. In this vision, the platform is a service and software developers code to it. The platform builds, deploys, and monitors your code. It balances load among your tasks, provisioning more resources when you need them. Your structured storage is provisioned according to your service needs, storage quantities, access patterns, and latency/availability requirements. Your unstructured storage is scalable and essentially infinite (or at least capable of tens of petabytes of capacity without significant planning). And when the infrastructure fails, it notifies you in a sensible, configurable, and correlated way. When the fault is with your code, debugging tools point to the problem. When the fault is with the infrastructure, mitigation is automatic.

We should be demanding more from our infrastructure providers than a quickly provisioned VM and some dashboards. We should be demanding a glorious future where we focus on interesting problems that really cannot be solved with software yet and rely on a platform to do the rest. When we get there, I'll be leading the charge towards the couch in the corner where we can sip bourbon and eat bon-bons. The extent to which DevOps encourages us to be satisfied with where we are is the extent to which DevOps and I part ways. I like what it's done so far, but I'm not satisfied.

Why You Probably Shouldn't Care About Operations Research

As the industry has been trying to figure out where to go, some people have suggested for the past few years that operations research is a really relevant source of academic thinking related to software operations. This is a field of study mostly focused on optimizing the output of a fixed set of constraints (traditionally a factory). The intent is to accurately model the set of constraints to understand where and why bottlenecks occur in the output. This facilitates intervention in the process design to improve output and efficiency.

Operations research is appealingly congruent to what we do in systems infrastructure. We try to build and maintain a set of infrastructure, with the intent of processing data with the minimum hardware and people, all the while maintaining service, availability, and performance within acceptable levels. Given a set of requirements ($x TiB of data at rest, $y queries per second of $z CPU cost per query on average), we should be able to provide the infrastructure specification that meets those requirements.

The problem arises as soon as you assume that the substrate below is fixed. It is not. It is software. Your network is software. Your chip's instruction set is software. Your OS is software. Your RPC marshalling/unmarshalling system is software. Your cluster file system is software. When it's convenient to think of each of these things as fixed, it is of course reasonable to do so. The constraints on the software stack, however, are massively less than the constraints on the machines in a factory. Your services infrastructure is exactly like a factory where a 3D printer can create new automation for each step at the same rate as you can specify and design it. We will have such factories, and when we do they will look much more like cloud infrastructure than cloud infrastructure looks like factories.

What's Next?

Let's back up for a second and review. Are operations and system administration already dead?

Not really. In some ways, the early part of this piece of writing is more exaggerated polemic than literal description. Our software and infrastructure are woefully incapable of actually handling a PostOps (or NoOps) world. For some time to come, the traditional sysadmin will continue to reinstall OSes, update configurations, troubleshoot network device drivers, specify new hardware, and reboot Windows boxes. But the writing is on the wall [2]. The computers are getting better and the tasks are getting more interesting. Most of these jobs will, and should, go away, to be replaced by a world with better, distributed infrastructure and more interesting jobs.

When that better, distributed infrastructure arrives, there are two things you need to know:

1. There will still be some operations to be done.
2. Operations will be done with software.

The wonderful, distributed self-regulating system of the future will not be stable and self-adjusting forever. It will encounter faults. Humans will have to intervene. When they do, operational sensibilities and approaches will be required to troubleshoot and resolve problems. So some amount of reactive, operational work endures. The scale and complexity will be much higher and the percentage of operational work will be much lower. You are definitely going to have to read, modify, and write software. If you're any kind of sysadmin, you already know how to code, but you may think you don't. You're already better than 50% of the people out there who are professionally employed as software engineers. Just get better than that.

As is usually the case, the future belongs to those who build it. So let us do that together—starting now.

References

[1] Adrian Cockcroft on NoOps: http://perfcap.blogspot.com/2012/03/ops-devops-and-noops-at-netflix.html.

[2] Todd Underwood, "PostOps: A Non-Surgical Tale of Software, Fragility, and Reliability," 27th Large Installation System Administration Conference (LISA '13): https://www.usenix.org/conference/lisa13/technical-sessions/plenary/underwood.

usenix
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

Publish and Present Your Work at USENIX Conferences

The program committees of the following conferences are seeking submissions. CiteSeer ranks the USENIX Conference Proceedings among the the top ten highest-impact publication venues for computer science.

Get more details about each of these Calls for Papers and Participation at www.usenix.org/cfp.

JETS* Volume 2, Number 3: *USENIX Journal of Election and Technology and Systems
JETS is a new hybrid journal/conference, in which papers will have a journal-style reviewing process and online-only publication. Accepted papers for Volume 2, Numbers 1–3, will be presented at EVT/WOTE '14, which takes place August 18–19, 2014.

CSET '14: 7th Workshop on Cyber Security Experimentation and Test
Co-located with the USENIX Security '14
August 18, 2014, San Diego, CA
Submissions due: April 25, 2014, 11:59 p.m. EDT
CSET invites submissions on the science of cyber security evaluation, as well as experimentation, measurement, metrics, data, and simulations as those subjects relate to computer and network security and privacy.

3GSE '14: 2014 USENIX Summit on Gaming, Games, and Gamification in Security Education
Co-located with the USENIX Security '14
August 18, 2014, San Diego, CA
3GSE is designed to bring together educators and game designers working in the growing field of digital games, non-digital games, pervasive games, gamification, contests, and competitions for computer security education. The summit will attempt to represent, through invited talks, panels, and demonstrations, a variety of approaches and issues related to using games for security education.

FOCI '14: 4th USENIX Workshop on Free and Open Communications on the Internet
Co-located with the USENIX Security '14
August 18, 2014, San Diego, CA
Submissions due May 13, 2014, 11:59 p.m. PDT

HealthTech '14: 2014 USENIX Summit on Health Information Technologies
Safety, Security, Privacy, and Interoperability of Health Information Technologies
Co-located with the USENIX Security '14
August 19, 2014, San Diego, CA
Submissions due May 9, 2014, 11:59 p.m. PDT

LISA '14: 28th Large Installation System Administration Conference
November 9–14, 2014, Seattle, WA
Submissions due: April 14, 2014, 11:59 p.m. PDT
If you're an IT operations professional, site-reliability engineer, system administrator, architect, software engineer, researcher, or otherwise involved in ensuring that IT services are effectively delivered to others—this is your conference, and we'd love to have you here.

OSDI '14: 11th USENIX Symposium on Operating Systems Design and Implementation
October 6–8, 2014, Broomfield, CO
Abstract registration due: April 24, 2014, 9:00 p.m. PDT
OSDI brings together professionals from academic and industrial backgrounds in what has become a premier forum for discussing the design, implementation, and implications of systems software.

SYSADMIN

Interview with John Looney

RIK FARROW

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

John Looney graduated from the School of Computer Applications at Dublin City University and specialized in supporting high-end commercial UNIX environments. He ran LinuxWorld Dublin in 2000 and, while at Hosting365, he built Ireland's largest shared hosting infrastructure for 30,000 customers on a shoestring budget. John is a site reliability engineer responsible for Google's cluster infrastructure: initially the cluster fabric, GFS and Chubby, and more recently the datacenter automation and remote access technologies. He has built a five-month, full-time graduate program to take junior engineers and retrain them to take the pager for Google.com. John is on the Computing Committee for Engineers Ireland. looney@google.com

During LISA '13, John Looney taught something he called "SRE Classroom: Non-Abstract Large System Design for Sysadmins" [1]. I had little idea what that was, and I was teaching on the same day so I couldn't drop in. But the description sounded too good to be true.

John had gathered ten fellow Google SREs to mentor people during class exercises, and the result was an experience that really had people talking. The class covered the design of large distributed systems, with exercises focused on building a detailed design. The designs are not specific to Google products but general enough for any large-scale system.

I was intrigued by the success of the tutorial, and wanted to learn more about where this class evolved. For me, John's explanation of what it's like to work at scale is fascinating and eye-opening.

Rik: Tell me about how you got involved with working with *nix and clusters.

John: In college, I loved messing around with the commercial UNIX workstations we had and, with a friend, secretly installed Linux on all the university Windows desktops. We repurposed one desktop with a broken monitor as server and covertly offered a Linux network to the students for their projects—we ran MySQL, NFS, etc. on Pentium 60s with VGA cards—it was so much more pleasant to use than SPARC ELCs with monochrome screens.

I did some Solaris support, various training and consulting services for Sun Microsystems after college and realized that though there were many training resources for Solaris sysadmins, there were none for Linux. So I started the Irish Linux User Group and arranged free community-led training programs every month. I made lots of great friends and learned so much. I got entangled in a few hilariously doomed startups, and eventually ran the network and systems of a Web hosting company with explosive growth for a few years. We never had much money, so had great fun building our own routers and servers with whatever hardware we could find.

This gave me the bug for supporting big systems, and I lucked out and joined Google's "Clusterops" team in 2005. At the time, we were responsible for the Google File System and all low-level cluster functionality.

Rik: My early sysadmin experiences did include Sun workstations, but we were still happy to have any network at all—UUCP over serial cables. So my experience stops well short of understanding what it's like administering clusters of servers. Could you tell us more about that?

John: In the Web hosting company, we had ~15 racks of equipment, most of which I'd installed myself from CD. When one failed, I knew which customer was affected, and usually exactly how to fix it. My first day in Google couldn't have been more different. "Here is a list of 18,000 broken machines. See how many you can get back serving, and if you think it's hardware, bounce it onto the hardware guys." My debugging skills needed to evolve. In the old days, I used to ssh in, poke around, work out what was wrong, write a small script to fix it, then run it on all other similarly broken machines.

A problem could be "Kernel X on platform Y in network Z emits 100 kernel log messages an hour filling the disk." The workaround might be "Truncate /var/log/messages on all affected machines with a cronjob." The permanent fix would be "File a bug with kernel team and get that kernel pushed out to affected machines once it passes testing."

In my Web-hosting days, that was a five-minute fire and forget shell command. In Google, shepherding that process could take a week…but it was worth it, if it was affecting 1000 machines; it could take more time if the buggy kernel was rolled out to all machines. Running clusters makes you think much more deeply about root causes and how to resolve issues permanently.

We have some very smart upper management in Google who realized that taking many weeks to turn up new clusters, by assigning new engineers the task, was a colossal waste of money. They told us to do the impossible—"Go work out how to turn up a cluster in a week. And the deadline is three months, because we have a half-dozen new clusters coming online that week."

We quickly built a technically beautiful system, based on Python unit test frameworks, to do cluster turn-ups. First, we wrote unit tests to verify that every aspect of a cluster was configured according to best practices. One unit test for "Does the DNS server IP respond to DNS queries?" and another for "Does it have an IP for its own address?" etc. There were thousands of tests that had to be written for everything from "Did anyone make a typo in the cluster definition database?" to "Does the Web search pass a load test?"

When a unit test failed, we had the automation framework run code that "repairs" the service, given that specific test failure: "DNS isn't listening on port 53…so go install the DNS package on machines reserved for DNS." "DNS is listening, but doesn't resolve correctly…execute code to push the latest configuration files to the DNS server."

It allowed geographically distributed teams to add tests and fixes for their own services, which could depend on each other (WebSearch depends on storage, which depends on Compute, which depends on Chubby, which depends on DNS). By the deadline date, we had a system that could execute months of manual, error-prone work in three hours. This saved a massive amount of money, but also meant we had up-to-date code that documented exactly how to turn up clusters, and could spot and repair configuration problems in running production systems. It was the application of engineering to an operations function, the definition of SRE.

Rik: You certainly make it clear that managing clusters is not like the system administration that I used to do. While there is still problem solving, automation becomes an important part of the solution.

What are some of the things that people interested in managing clusters can learn that will help them?

John: Try to keep abreast of the state of the art. I'm fortunate that in Google, if I have a problem I can usually find ten teams who have had that problem first and have taken a stab at solving it once or twice.

In the outside world, it's common to think your specific problem is unique, and the existing tools don't work. You are probably wrong :). Conferences and user groups are good places to find experienced folks who can talk you out of trying to build something new, but have ideas on how to customize and contribute to an existing system that will solve your problem.

Of course, that means that sysadmins need to be able to code in the common systems and automation languages—C, Python, Java, and perhaps one of the less common systems. "I only know AWK and Perl" isn't good enough anymore. If you are on a very small site, with few coders, it's really important to network and use the free software community as a resource to allow you develop software engineering skills sysadmins need—writing code that can be tested, is efficient, and that others can maintain.

I'm a big believer in "whole stack" knowledge for a sysadmin. You should be able to talk about Arduino-level electronics, the latest CPU and RAM designs, and the physics of hard disk manufacture. Books like *The Elements of Computing Systems: Building a Modern Computer from First Principles* (www.nand2tetris.org) are wonderful for this "low-level" knowledge. You should know your way around the operating system kernel and be able to write simple device drivers (even if it's just to talk to an Arduino over serial or something).

One of the ways sysadmins can surprise and intimidate software engineers is by being able to take a broken system they've never seen, and diagnose and suggest a fix. You get good at this by practicing—downloading random free software packages, trying them out, shouting aloud at dumb config systems, but making it work anyway. Go a step further—suggest or make changes to the software, and get the changes into the upstream code. Ask the author why it was done that way in the past. The modern sysadmin needs to be a software archaeologist when needed.

There are some powerful configuration systems out there these days. It seems most people use them in a very procedural way—"This is the directory for Web server configs, one file per machine, this is the directory for SSH daemons." Learn how your whole network looks from above and build models (in a text file, SQL database, etc.) that describe everything about your system… then write tools that take that "model" and build the configs for you. Cluster admins shouldn't edit text files, unless they configure systems that configure systems that write text files :).

Lastly, I'd recommend that they try to think bigger. We often try to solve the problem in front of us the way we solved it last year. If you are in an environment that tolerates failure (aka innovation). At LISA last year, I had great fun getting 70 sysadmins to try to design a system, like imgur.com, in enough detail to work out how much it would cost to buy the hardware. By pushing our limits, in design, capacity planning, architecture, and presentation skills, we can learn a lot about ourselves.

Rik: That was a very popular class. Can you tell us more about that class and how it was run?

John: The interview process for SRE is pretty tough, as we grill people on networking, UNIX, software engineering, project management, and other skills we might find useful. The interview that most people struggle with is "non-abstract large-scale design." It's probably because it's not something everyone does. If you do well in that interview, it's a good sign you will make a good SRE. If you do poorly…we can't draw conclusions.

Some SREs in London decided to run an experiment—locate SRE job applicants who were potential SREs, but were too inexperienced to succeed at a large-scale design interview. Invite them in for a one-day class on design, explain how such a skill would be applied in the real world, and if they felt up to it, have them interview for an SRE job the next day.

It's one thing to be able to design, but it's also important to be able to communicate that design to people. So they designed a class that would introduce the concepts of design, and allow people to practice them, and discuss them in groups. We discussed how to approach an SLA-based design, some common facets of large-scale systems, failure modes of distributed systems, and monitoring such systems. We interleaved the classes with discussions on design in small groups, each of which was led by an experienced Googler.

We've now run this class in a few offices, and helped a good number of people to get hired by Google (success!). I was asked to help with a tutorial at LISA and decided that I'd love to try to do a similar event. I panicked a little when I found out we had nearly 70 attendees signed up—in Google we had one "mentor" per five attendees. I made an impassioned plea to the Google SRE team, and ten volunteers traveled to Washington and gave up their Sunday to share their love of crazy big systems.

I think if you assume that your attendees are up for a real challenge, are willing to have their brains turned inside-out with some really good material, and are willing to try something they've never done before, you can have a lot of fun.

Rik: That's quite a story. I really had no idea that members of Google SRE teams would give up part of their weekend to mentor students.

USENIX hopes to find other tutorials that will help people learn what it takes to work at scale. Do you have any ideas for other class topics that might be useful? Other people have suggested data analysis and release engineering.

John: Release engineering is vital. At scale, you are likely running custom software, so you can't assume someone else has tested everything for you. It's also not acceptable to have scheduled downtime, so building and testing software that can be incrementally drained and upgraded without a problem isn't trivial.

Data analytics could be useful; it could also be useful to learn how to take "production quality" software and instrument it. Add the equivalent of an Apache "status" page that tells all the incoming and outgoing requests, latencies, RAM usage, etc. Something that can be aggregated by your monitoring system later.

I'd also love to build a distributed debugging class, but I fear debugging is so domain specific it wouldn't be useful—or easy to run. We need to optimize for people's time and energy. If you'd like to be involved, please contact LISA's Program Committee for 2014.

Reference

[1] SRE Classroom: https://www.usenix.org/conference/lisa13/training-program/full-training-program#S2.

usenix THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

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

How to Be a Better System Administrator and Then Something Else

ELIZABETH ZWICKY

After years as a system administrator, Elizabeth has experienced an almost full recovery but happily has maintained her ability to intimidate the help desk into providing sensible answers. Zwicky@otoh.org

I once was a system administrator. Now I'm not. This caused somebody to think I might have useful career advice, which is absurd if you look at my career trajectory, which looks like a ball of string after the kitten got to it. In fact I am, as the saying has it, a highly paid computer professional and have been all along, and most of the things that have caused people to give me other jobs are skills I honed as a system administrator.

There is one thing which will make no difference to your performance as a system administrator but will make getting another job in technology vastly easier, and that is a college degree, preferably in computer science or a related field. (No, I don't know what "a related field" is, except I have evidence that if you make this claim and turn out not to have had any classes involving a computer, people will fire you.) This is a fact about the job market, and its actual wisdom or lack thereof is irrelevant. If you can get a college degree, do so; and, if involving computing in it is still a reasonable option for you, drag some computers into it.

Then learn to code. If you do this by solving problems at work, it will make you a better system administrator. The process will also teach you things about coding that most courses and books don't teach well. If you ask my colleagues, they will tell you, with affectionate condescension, that I "don't really code." You might think, based on that, that I rarely write code (in fact, I do so almost every day), or that I never modify production systems (in fact, I am one of the people called upon to write changes directly into major customer-facing systems when that needs to be done on the fly), or that I don't use the main programming languages we use (this is almost true, but I do review and fix code in all of them).

In fact, what they mean is that I don't spend all day writing code, and that I almost never produce an entire application myself.

I really learned to code as a system administrator. I have a computer science degree, which as I mentioned above is a valuable piece of paper, and it certainly taught me what variables are, how command structures work, and why $O(n^2)$ is bad. However, it is the single least relevant piece of my computing experience to my work life—well, behind my teenage days trying to make BASIC play Animal better. Because my real learning was on system administration tasks, it leaned heavily on regular expressions, on reading and porting other people's code in whatever language they chose, and on building utility chains. This means that if you want an application written from the ground up, I am nearly useless at actually producing the code. On the other hand, if you want somebody to do that with moderate competence, you can probably find three at the nearest coffee shop. All of them will blanch if you ask them to write a regular expression, and if they do write it, it will contain at least one unnecessary and dangerous ".*".

So embrace the funkiness of writing the code that your situation needs. It may not look like coding in school, or even coding as a full-time programmer, but it will give you the ability to round out a programming team.

And embrace the rest of the limitations you may face. You have to debug third-party software with no source code? Great! You will never be any of the people who took my bug reports and closed them because "the code doesn't do that," which is a phrase liable to inspire me to violence. I really don't care whether the code does it or the pack of trained gerbils living under the floor does it. If I reported it as a bug, I want it to stop already. You will also not be the person who never uses a debugger because you can always just put in some print statements, right? (No, you can't.)

Learning to debug distributed systems written by other people will also give you a head start on being data-driven, because when you are trying to figure out what happened on somebody else's computer system, you don't have a lot of choice except to start digging data out of the system. What's in the logs? What files got changed? Programmers who can work in a change-and-try system don't get forced to that mode of thought. Admittedly, only the good system administrators do. Debugging is a theme, and excellence at system administration is what turns into transferrable skills.

What else goes with that theme?

- **Understand the technology you work with; in particular, know how networks and file systems work.** For some reason, computer science degrees manage to get people to recite facts about compilers, operating systems, and maybe even databases, but most of them are not entirely sure the Internet is not made of tubes.
- **Learn to be good in emergencies.** System administrators get called in the middle of the night when things are broken and people are screaming. This builds important life skills for times you don't always have a pager. In particular, it teaches you to be very, very careful, because everybody is stupid in the middle of the night and you need to be able to recover. I don't edit important files in place. I don't do it much in broad daylight when I'm relaxed and caffeinated, but I don't do it at all when the stakes are high—I don't trust software and I don't trust myself. So I make a copy, and a backup copy, and then I edit the copy, and then I copy the edited copy to the original location and see what happens. You can argue about the details of this process—I'm sure you're dying to inform me that all your files are in source control, which is even better—but the point is that somebody has already paid to have this caution instilled in me, which is a win for all my recent employers.
- **Learn to be customer-oriented.** I don't care if you call the people who use your systems "users," but you need to be able to think about what every change means to them. Again, this is a skill that differentiates senior people from junior people, regardless of where you are, and it's one you can learn in stark and dramatic terms in system administration, when your users are often senior to you and capable of physically yelling at you.
- **Learn capacity planning and performance tuning.** This is full of useful mathematics; if you don't figure out why the mean is not the interesting average when you start looking at how users use disk space, or send email messages, or how your network is utilized, you are never going to understand numbers. Building big production systems is dependent on an ability to think about the numbers in ways they don't teach in school, based on the number of mid-level programmers who, when asked questions like "Will I be able to do that in my 100-millisecond budget for this operation?" or "How much space will that data take up?" give answers like "It's really fast" and "It shouldn't be a problem in the cloud." (Hint: I didn't ask idly. If your company built cloud technology, it is because it has problems that require a cloud, instead of problems that are trivial with a cloud.)
- **Learn about security.** When you're responsible for the system, you're responsible for protecting it. This is the time to learn about threat models and attack surfaces. Engineering a system to resist an active attacker is a different skill from engineering one that is resilient to pure error. And those debugging skills I mentioned earlier? They're security skills as well, because one of the chief jobs of security and abuse teams is distinguishing security and abuse issues from other issues. Not only are programmers bad at security, but carefully purpose-trained security people spend years seeing bad guys behind every crash. Converted system administrators have a finely honed respect for the basic malevolence of the universe.
- **Learn to pick up new technologies.** Again, as a system administrator, you are often pretty much stuck adapting yourself to the choices of your management and the people you support. Embrace and extend; get good at figuring out the rudiments of whatever it is they are doing. When I am hiring new programmers, my first choice is one who knows one of the languages we use a lot really well but is happy to learn others. My second choice is somebody who doesn't really care deeply what languages we use a lot but knows a little bit about a couple of our languages. The person who has a really strong opinion about what languages we ought to use is not very high up on my list, no matter how good they are at the languages they know, and even if they think our current choice is right, because we slowly but surely change our language balance over time.

There are dozens of other things I could add. I've mentioned security, programming, and network engineering, but system administration could also lead you into release engineering, or technical writing, or project management, or database administration, all skills that are careers on their own but which system administrators are often expected to do on their own. I hope I have convinced you—and helped you to convince future employers—that system administration is not only valuable work in itself, but also a solid platform for doing other work if that's of interest.

Accelerating the Path from Dev to DevOps

DINAH MCNUTT

Dinah McNutt is a release engineer at Google. She has a master's degree in mechanical engineering from MIT and has worked in the fields of system administration and release engineering for more than 25 years. She's written articles for numerous publications and has spoken at technical conferences (including chairing LISA VIII). mcnutt@google.com

My first lesson in release engineering occurred more than 20 years ago. I was working for a start-up company, and we discovered that we could not reproduce the build we had shipped to customers. This meant we could not send out a patch for this release and our only solution was to force all our customers to upgrade to the new version. I was not directly involved in the events that got us into this situation, but I certainly learned from it.

I've spent most of my career in a system administration role at start-up companies and have learned a lot about software development and releasing products. Twelve years ago, I fell into a release engineer position when the company I was working for needed someone to do the work, and I discovered I loved it. All the skills that made me a good system administrator (problem solving, attention to detail, etc.) were directly applicable.

What Is Release Engineering?

Release engineering (or releng, pronounced "rel-eng" with a soft g) is like the old story of the blindfolded people and the elephant. You may get a different answer depending on whom you ask. But, because this is my article, I get to describe my elephant.

In a perfect world, a release process looks like:

- Compile
- Test
- Package
- Release

A real release process looks more like what is shown in Figure 1.

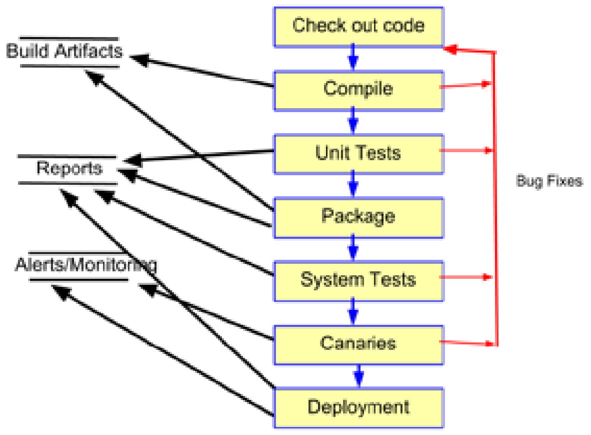


Figure 1: A real world release process

I'm not going to go into the details of Figure 1 because the point is to show that most release processes are complicated. However, here are some terms from the figure that you might not be familiar with:

- *Build artifacts.* By-products of the release process (i.e., log files, test results, and packaged binaries). Basically, it's everything you want to save from the release process.
- *Canaries.* Testing new software on a small number of machines or with a small number of users.

As the tagline to this article says, releng accelerates the path from development to DevOps by bringing order to the chaos shown in Figure 1. How do we do that?

Building Blocks

My eyes usually glaze over when I hear people talk about velocity, agility, delivery, auditing, etc. Those concepts are the attributes and results of good releng practices but are not where I like to start when I talk about releng.

Instead, here are the things I care about:

- *Release engineering from the beginning*. Releng is often an afterthought. Companies should budget for releng resources at the beginning of the product cycle. It's cheaper to put good practices and process in place early rather than have to retrofit them later. It is essential that the developers and release engineers (also called releng) work together. The releng need to understand the intention of how the code should be built and deployed. The developers shouldn't build and "throw the results over the fence" to be handled later by the releng. It's OK to outsource the implementation of your releng processes, but keep the ownership and design in-house.
- *Source code management*. Everything needs to be checked into a source code repository. It's not just about code. Configuration files, build scripts, installation scripts, and anything that can be edited and versioned should be in your SCM. You need to have branching/merging strategies and choose an SCM system that makes these tasks easy. I personally think you should have different strategies for ASCII and non-ASCII files (like binaries). I am not a fan of storing binaries with source code, but I do think it is reasonable to have separate repositories for those types of files. (This is one of those topics in which even members on the same releng team do not agree!)
- *Build configuration files*. The releng should work closely with the developers to create configuration files for compiling, assembling, and building that are consistent and forward thinking (e.g., portable and low-maintenance). Do they support multiple architectures? Do you have to edit hundreds of files if you need to change compile flags? Most developers hate dealing with build configuration files, but a releng can make their lives easier by taking the lead in this task.
- *Automated build system.* You need to be able to build quickly and on-demand. The build process needs to be fully automated and do things like run tests, packaging, and even deployment. Your build system should support continuous and periodic (e.g., nightly) builds. A continuous build is usually triggered by code submissions. Frequent builds can reduce costs through early identification (and correction) of bugs.
- *Identification mechanism*. There should be a build ID that uniquely identifies what is contained in a package or product, and each build artifact needs to be tagged with this build ID.
- *Packaging*. Use a package manager. (As I have said repeatedly, tar is not a package manager.) You have to plan ahead for upgrades, handling multiple architectures, dependencies, uninstalls, versioning, etc. The metadata associated with a package should allow you to determine how the binaries were built and correlate the versions to the original source code in the source code repository.
- *Reporting/Auditing*. What was built when? Were there any failures or warnings? What versions of the products are running on your servers? Logs, logs, and more logs. (We like logs.)
- *Best practices*. What compile flags should you use? How are you versioning your binaries so you can identify them? Are you using a consistent package layout? Can you enforce policies and procedures?
- *Control of the build environment.* Do your tools allow you to put policies in place to ensure consistency? If two people attempt the same build, do they get identical results? Do you build from scratch each time or support incremental builds? How do you configure your build environments so you can migrate your tool base to newer versions yet still be able to support and build older versions of your code?

I've described the building blocks of release engineering. Through effective use of these building blocks, you can

- Continuously deliver new products (e.g., high-velocity)
- Identify bugs early through automated builds and testing
- Understand dependencies and differences between different products
- Repeatedly create a specific version of a product
- Guarantee hermetic build processes
- Enforce policy and procedures (this is a hard one—you at least need to be aware of violations and exceptions)

Sub-disciplines within Releng

Releng is an evolving discipline. It's going to be exciting to see how it changes over the next few years. At many companies, releng is just one of several hats worn. At LISA '13, I held a Birds-of-a-Feather session on release engineering. Several people attended who have a dual role as system administrator and release engineer. Because I come from a system administration background, that makes perfect sense to me!

However, at a large company like Google, we are starting to see specialization within the releng team that is dictated by product area and personal preference. Here are the sub-disciplines I have identified:

- *Tools development.* Extending and customizing our proprietary build tools; developing stand-alone applications to provide reporting on everything from build status to statistics about build configuration files.
- *Audit compliance.* This is no one's favorite task, but the Sarbanes-Oxley Act of 2002 dictates that controls must be put into place for applications that handle financial information. The controls include (but are not limited to):
  - Verifying all code that is under scope for SOX has undergone a code review (separation of duties)
  - Verifying the person who writes the code must not also own the build and deployment processes (separation of duties)
  - Embedding a unique ID that can tie the binary to the build that produced it (version verification)
  - Using a package manager that supports signatures so the package can be signed by the user who built it (builder and version verification)

  Release engineers work with developers and internal auditors to ensure that appropriate controls and separation of duties are in place.
- *Metrics.* We have several projects that provide releng-related metrics (build frequency, test failures, deployment time, etc.). Some of these tools were developed by members of the releng team.
- *Automation and execution.* We have proprietary continuous-build tools, which are used to automate the release process. Release frequency varies widely (from hourly to yearly). Typically, customer-facing applications are released more frequently in order to get new features out as quickly as possible. Internal services are usually updated less frequently because infrastructure changes can be more expensive. However, with effort, release processes can be developed which support frequent, low-impact changes.
- *Consultation and support.* The releng team provides a suite of services to development teams, which range from consulting to complete automation and execution of the releases.
- *Source code repository management.* We have a dedicated team of software engineers and administrators who work on our source code management system, but many of the release engineers have in-depth knowledge of the system. We even have engineers who transferred from the source code repository team to a releng team!
- *Best practices.* This covers everything from compiler flags to build ID formats to which tasks are required to be executed during a build. Clear documentation makes it easy for development teams to focus on getting their projects set up and not have to make decisions about these things. It also gives us consistency in how our products are built and deployed.
- *Deployment.* Google has an army of Site Reliability Engineers (SREs) who are charged with deploying products and keeping google.com up and running. Many of the releng work closely with SREs to make sure we implement a release process that meets their requirements. I spend just as much time working with SREs as I do Software Engineers (SWEs). We develop strategies for canarying changes, pushing out new releases without interruption of services, and rolling back features that demonstrate problems.

What's Next?

Here is what I expect to see over the next few years in the field of release engineering:

- More vendors entering the space (particularly cloud-based solutions). Look for productization around open source software (e.g., Git) and tools that will offer end-to-end release engineering solutions. The latter will probably be achieved through partnerships between vendors.
- Fuzzy lines between configuration management and release engineering (my prediction is that they will evolve into a single discipline)
- Standards organizations—ISO standards for releasing highly reliable software, SOX compliance standards, etc.
- Industry-standard job ladders
- College curriculums
- Industry-accepted best practices
- Industry-accepted metrics

I am excited to be chairing the upcoming URES '14 (USENIX Release Engineering Summit). As we are starting to put the conference program together, we wanted to be able to easily explain what release engineering is and why it is important (and timely) for USENIX to sponsor a summit on this topic. I hope this article has been a good introduction to release engineering and that my personal experiences have been educational. May all your software come from reliable, reproducible processes!

Interview with Tom Hatch

RIK FARROW

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

Tom is the creator and principal architect of SaltStack. His years of experience as principal cloud architect for Beyond Oblivion, software engineer for Applied Signal Technology, and system admin for Backcountry.com provided real-world insight into requirements of the modern datacenter not met by existing tools. Tom's knowledge and hands-on experience with dozens of new and old infrastructure management technologies helped to establish the vision for Salt. Today, Tom is one of the most active contributors in the open source community. For his work on Salt, Tom received the Black Duck "Rookie of the Year" award in 2012 and was named to the GitHub Octoverse Top 10 list for the project with the highest number of unique contributors, rubbing shoulders with projects like Ruby on Rails and OpenStack.
thatch@saltstack.com

Rikki Endsley met Tom Hatch, CTO of SaltStack and creator of the Salt [1] open source project, during USENIX 2013 Federated ConferencesWeek. Rikki suggested I interview Tom and, when I found out that Salt uses ZeroMQ, my own curiosity was piqued.

ZeroMQ is one of a new breed of asynchronous messaging libraries. These libraries hide the complexity of developing your own queueing software. They also can use reliable multicast protocols, which can cut down on the amount of network traffic when many hosts will be receiving the same information. Message queueing also helps prevent overloading the server, a problem known as incast.

While Salt competes with more popular configuration management systems, like Puppet, Chef, and CFEngine, I thought that what makes it different also makes it important to take a deeper look at Salt.

Rik: There are lots of configuration management systems available today, but what caught my eye was the mention of Salt using ZeroMQ. Tell us about how Salt uses ZeroMQ and what the advantages of doing that are.

Tom: The big thing to remember about Salt is that it was first and foremost a remote execution system, and this is the reason I used ZeroMQ initially. ZeroMQ has been a great help with the configuration management system as well. The ability to have queued connections has allowed Salt to scale very well out of the box without modification and has been a foundation for Salt's flexibility.

The real benefit of ZeroMQ is that it has allowed us to do things where we have clean communication about anything in a deployment so that decisions can be made about configurations anywhere, which allows us to make a truly ad hoc distributed configuration management system. We are working on a UDP alternative to ZeroMQ to get past some of the limitations of ZeroMQ.

Rik: What specifically attracted you to using message queueing in the first place, instead of more traditional means of communication, like SSH over TCP?

Tom: Really, it was the PUB/SUB interface in ZeroMQ. PUB/SUB is very difficult to do with non-async systems and is very slow. ZeroMQ gave us an easy way to have a PUB/SUB interface that works well for remote execution. Beyond that, ZeroMQ has continually proven itself a great asset in communication with large groups of systems.

Salt does also ship with an optional SSH transport to have an "agentless" approach, but ZeroMQ is still substantially more scalable and faster.

To get more into the weeds, ZeroMQ's ability to queue connections and commands on the client side allows for very large numbers of systems to request information at once and all wait for the server end to be available rather than just bogging down the server. This is one of the key advantages in the Salt file server. This file server is built entirely on top of ZeroMQ from scratch and subsequently is capable of sending very large numbers of small files to large groups of systems with great efficiency.

Rik: You just mentioned the Salt file server. Is this how you store commands to be executed on clients? Or is this something else, like the general term for your server, which can interface with other CMDBs?

Tom: The Salt file server is just part of the Salt master. The master includes a fully functional file server and is a critical part of the CM system in Salt, because the salt minions (the agents) download the configuration management files from the master and compile them locally. The Salt file server can also be used to distribute large files like VM disk images. This is part of how Salt's cloud controller, salt-virt, works.

ZeroMQ is what makes serving files so scalable. It lets Salt minions download chunks of files and queue for their downloads, which keeps things humming and prevents overload on the master.

So the commands to be executed in the CM system are all stored in files (typically YAML files) and served via the Salt file system. Then the commands are compiled down to data structures on the minion and executed.

Salt has a lot of services in the master. For instance, the Pillar system is what allows the master to connect to external systems like CMDBs to get raw data which can be used when compiling the config management files into execution data.

Rik: I watched your presentation at UCMS '13 [2], and you mentioned the Pillar system. So Pillar can extract configuration information from existing systems, like Puppet or Chef, and then provide it to Salt minions? Do the minions use the original configuration agent, or are configuration actions performed natively by the minions?

Tom: Pillar is a system that allows for importing generic data, not just configuration data. It is minion-specific data that is generated on the master. So think of it more as a variable store that can be accessed from Salt's configuration management system.

Configuration management is performed natively in Salt by the minion. The great thing about Pillar is that it can be a top-level data store. Pillar makes it VERY easy to make generic configuration management formulas in Salt.

Let me try and sum up a few things since we are kind of jumping from topic to topic.

Salt is a remote execution platform that can execute generic commands on clients called minions. This generic command execution means that Salt is often used to orchestrate deployments that already use Puppet, and Wikipedia is a classic example.

But Salt has its own CM system which incorporates features not found in Puppet or Chef and ties directly into the remote execution system, allowing for many cross-communication routines to work—all over ZeroMQ.

Salt management is all about data, so the config management files that Salt uses, or formulas, can use Pillar (data generated on the master—optionally from external sources like CMDBs) or grain data (data generated on the minion, things like the OS version—kind of like Puppet's Facter [3]) as variables or to decide what routines should be executed.

So to answer your question, Pillar can pull data out of systems that Puppet uses, like Hiera, and reuse the data in its own configurations, or it can just call Puppet directly on the client. Or everything can be directly managed by Salt.

Rik: I find it interesting that Salt can handle data returned by the minion. Your UCMS presentation hinted at that. Getting configuration information back from the client, like Puppet's Facter, is certainly useful for configuration management. But because Salt is a remote execution system, I was wondering how it handles other information that might be returned by a client: for example, a failed compilation of a downloaded package because of missing library dependencies? How hard is it to set up Salt to handle other information, like my example, from a minion?

Tom: What you asked in the first question starts to hint at where Salt goes beyond configuration management. Since Salt is based on remote execution and events, any time something fails in a configuration management run, an event gets fired on the master. This means that the Salt Reactor can pick up the event and react to it. The reaction is arbitrary: it can reach out to the minion and try the install again, or it can react by destroying the virtual machine running the minion (assuming it is a VM, as an example) by communicating to the hypervisor, or to the cloud system, such as OpenStack, AWS, or any major cloud provider.

Also, since Salt's configuration management system is natively idempotent, it is easy to fire the configuration management to run again, either manually or via the Reactor.

References

[1] Salt: https://github.com/saltstack/salt.

[2] UCMS SaltStack talk: https://www.usenix.org/conference/ucms13/summit-program/presentation/Hatch.

[3] Puppet Facter: http://puppetlabs.com/facter.

The Case of the Clumsy Kernel

BRENDAN GREGG

Brendan Gregg is the lead performance engineer at Joyent, where he analyzes performance and scalability at any level of the software stack. He is the author of the book *Systems Performance* (Prentice Hall, 2013) and is the recipient of the USENIX 2013 LISA Award for Outstanding Achievement in System Administration. He was previously a performance lead and kernel engineer at Sun Microsystems, where he developed the ZFS L2ARC, and later Oracle. He has invented and developed performance analysis tools, which are included in multiple operating systems, and has recently developed performance visualizations for illumos and Linux kernel analysis. brendan.gregg@joyent.com

All benchmarks are wrong until proven otherwise. Benchmarking is an amazingly error-prone activity, with results commonly misinterpreted and wrong conclusions drawn. However, every now and then, a benchmark takes me by surprise and not only is correct but also identifies a legitimate issue. This is a story of debugging a benchmark.

A Joyent customer had benchmarked Node.js connection rates and found a competitor had five times higher throughput than we did. Because we're supposed to be the "High Performance Cloud," as well as the stewards of Node.js, this was more than a little embarrassing. I was asked to troubleshoot and determine what was happening: were the results misleading, or was there a real performance issue? We hoped that the problem was something simple, like the benchmark system hitting a CPU limit.

Our support staff had already begun collecting a problem statement, which included checking which software versions were used. This process can solve some issues immediately, without any hands-on analysis. The benchmark was Apache Bench (ab) [1], measuring the rate of HTTP connections to a simple Node.js program from 100 simulated clients. This was about as simple as it gets.

Some factors can make these investigations very hard. The worst I deal with involve networking between multiple hardware-virtualized guests, which means trekking between guest and host kernels via a hypervisor, and where those kernels are entirely different (Linux and illumos [2]). In this case, it was on a single host via localhost, and in an OS-virtualized guest. These factors took the physical network components and lower-level network stack completely off the investigation table and left only one kernel to study (illumos). This should be easy, I thought.

The USE Method

I created a server instance and ran the same benchmark that the customer had. Benchmarks like ab run as fast as they can and are usually limited by some systemic bottleneck. The utilization, saturation, and errors (USE) method is a good way to identify these bottlenecks [3], and I performed it while the benchmark was executing. The USE method involves checking physical system resources: CPUs, memory, disks, network, as well as resource controls. I discovered that CPUs, which I expected to be the limiter for this test, were largely idle and also were not driven near the cloud-imposed limit. The USE method often gives me quick and early wins, but not in this case.

By this point, I had run ab a few times and noticed that its results matched what the customer had seen: the connection rate averaged around 1000 per second. Because I don't trust anything any benchmark software tells me, ever, I looked for a second way to verify the result and to get more information about it. Running on SmartOS [4], I used "netstat -s 1" to print per-second summaries, which also shows per-second variation (on Linux, I would have used "sar -n TCP 1").

```
$ netstat -s 1 | grep ActiveOpen
    tcpActiveOpens  =728004  tcpPassiveOpens  =726547
    tcpActiveOpens  =   4939  tcpPassiveOpens  =   4939
    tcpActiveOpens  =   5849  tcpPassiveOpens  =   5798
    tcpActiveOpens  =   1341  tcpPassiveOpens  =   1292
    tcpActiveOpens  =   1006  tcpPassiveOpens  =   1008
    tcpActiveOpens  =    872  tcpPassiveOpens  =    870
    tcpActiveOpens  =    932  tcpPassiveOpens  =    932
    tcpActiveOpens  =    879  tcpPassiveOpens  =    879
[…]
```

The first line of output is the summary since boot, followed by per-second summaries. What caught my eye was the change in connection rate: starting around 5000 per second, and then slowing down after two seconds. This was not only a great lead, it also rang a bell. I remembered that this type of benchmarking can hit a problem involving TCP TIME_WAIT. This state occurs when the SYN packet heralding a new connection is misidentified as belonging to an old connection and so is dropped by the kernel. My test for this issue is to see how many connections are stuck in TIME_WAIT, and whether the client's ephemeral port range is exhausted—which causes every new connection to clash with an old one. I used netstat and saw that there were only around 11,000 connections from a possible range of about 33,000. So much for that theory.

What else might be happening after two seconds? I drew a blank.

Thread State Analysis

My other go-to methodology is the thread state analysis (TSA) method [5], where thread run time is divided into states, and then you investigate the largest states. On Linux, I'd perform this using tools including pidstat. On SmartOS, I used prstat [6]. When ab was running at 5000 connections per second, this showed that a single thread in node (the runtime process for Node.js) was on-CPU 100% of the time. This was the kind of CPU limit I had expected to hit. When ab slowed down, prstat showed:

```
$ prstat -mLc 1
  PID USERNAME  USR SYS  TRP  TFL  DFL  LCK  SLP  LAT  VCX  ICX   SCL SIG  PROCESS/LWPID
63037 root       15 3.6  0.0  0.0  0.0  0.0  81   0.2  268 26    8K   0  node/1
12927 root      2.4 8.3  0.0  0.0  0.0  0.0  89   0.7   1K 42   16K   0  ab/1
[...]
```

The node thread was now spending 81% of its time in the sleep (SLP) state, meaning the thread is blocked waiting for some event to complete, typically I/O. Were this performed between two remote hosts on a network, I would guess that it was waiting for network packet latency. But this was a localhost test!

One way to investigate the sleep state is to trace system calls and their latency. I may find that the sleep time is during read(), or accept(), or recv(), and I can investigate each accordingly. On Linux, I'd use one of the (in-development) tracing tools, which include ktap, SystemTap, dtrace4linux, and perf, or, if I didn't mind the overhead, strace. Because this was SmartOS, I used DTrace and quickly found that the sleep time was in the portfs() system call. The user-level stacks that led to portfs() told me little: the threads were polling for events.

portfs() is part of the event ports implementation, which has a similar role to epoll on Linux: an efficient way to wait on multiple file descriptors. Being blocked on portfs() meant we were blocked on something else, but we didn't know what, and it would be a bit of work just to dig out the right file descriptors.

This was looking like a dead end. Imagine you have a thread blocked polling on portfs(), or on Linux, epoll_wait(). What do you investigate next?

Tracers can lead you to think in a thread-oriented manner. If the thread time between A and B is of interest, then you look for events that happened between A and B for that thread, and measure their relative times. But what if the thread does nothing between A and B, as was the case here? Time has been spent on something else in the kernel—something mysterious and likely involving other threads. There is no easy way to correlate this activity, let alone know what activity or threads to trace to in the first place.

Walking the Wakeups

There is a way, however, and it's one that I've been using more and more of late. My approach is to trace the kernel as it performs wakeups: where one thread wakes up another sleeping thread. This provides correlation and context: the stack trace of the waker.

I used cv_wakeup_slow.d [7], modified to trace node processes. This is a DTrace script I wrote earlier, which shows the stack trace of the threads that woke up a specified target (the cv is for conditional variable, which is how the sleep and wake up is implemented by the kernel). I ran it with a 10 ms threshold and caught:

```
# ./cv_wakeup_slow.d 10
[…]
 23  12326    sleepq_wakeone_chan:wakeup 63037 1 0 0 sched… 46 ms
           genunix`cv_signal+0xa0
           genunix`port_send_event+0x131
           genunix`pollwakeup+0x86
           sockfs`so_notify_newconn+0x81
           sockfs`so_newconn+0x159
           ip`tcp_newconn_notify+0x198
           ip`tcp_input_data+0x1b4a
           ip`squeue_drain+0x2fa
           ip`squeue_enter+0x28e
           ip`tcp_input_listener+0x1197
           ip`squeue_drain+0x2fa
           ip`squeue_enter+0x28e
           ip`ip_fanout_v4+0xc7c
           ip`ire_send_local_v4+0x1d1
           ip`conn_ip_output+0x190
           ip`tcp_send_data+0x59
           ip`tcp_timer+0x6b2
           ip`tcp_timer_handler+0x3e
           ip`squeue_drain+0x2fa
           ip`squeue_worker+0xc0
```

This stack trace shows a thread slept for 46 ms and was woken up by a TCP packet. Interpreting latency depends on application needs and expectations for the target. In this case, I was expecting the benchmark to stay on-CPU as much as possible, so any non-zero sleep time was worth investigating. My choice of a 10-ms threshold was intended to filter out noise from occasional systemic perturbations, such as interrupts preempting the benchmark. These perturbations should be fast (sub-millisecond), and unlikely to cause the 81% sleep time I saw earlier. But if a 10-ms threshold came up empty-handed, I'd reduce that to 1 ms, and, if need be, to 0 ms so I could see all events.

Looking down the stack shows tcp_timer() calling tcp_send_data(). Huh? I took a quick look at the tcp_timer() code, which largely handles TCP retransmission. Retransmits?

I checked the retransmission rate compared to the connection rate using "netstat -s 1" (on Linux, use "sar -n TCP -n ETCP 1"). When the connection rate from ab was high, the retransmit rate was zero. But, when retransmits began to occur, the connection rate slowed down. This correlation matched what I'd found with the wakeup tracing: the benchmark was getting blocked waiting on retransmits.

But…retransmits? Over localhost? How is this possible?

Retransmits can be a sign of a poor physical network, including bad wiring, cables not plugged in properly, an overloaded network, TCP incast, and other reasons. But this was localhost, where the kernel is passing packets to itself, with no networks (reliable or otherwise) involved. I mentioned this to a colleague, Robert, and we were amused by the mental image of a clumsy kernel, dropping packets as it passed them from one hand to the other.

We did remember some legitimate reasons why a kernel might drop packets (firewalls, out of memory, etc.), which could lead to retransmits. And there was always the possibility of bugs. It wouldn't be the first localhost bug I've seen, and I shuddered at the thought of finding another.

I noticed something else about the retransmit rates: they seemed to hit a ceiling of 100 per second. ab was simulating 100 clients, and the TCP retransmit interval was one second. This fit: each client could do at most one retransmit per second, because it would then spend an entire second blocked on the retransmit. As an experiment, I set the ab client count to 333, and, sure enough, the retransmits moved to a ceiling of 333 per second.

I used another DTrace script I had written earlier [8] to trace retransmit packets and show their TCP state. This script quickly tells me whether the retransmitted packets were from an established connection, or from a different stage of a TCP session. Such kernel state information is not visible on the wire (or "wire" in quotes, as this is localhost), so it cannot be observed directly using packet sniffers. I hesitate to use packet sniffers for this kind of investigation anyway, because their overheads can change the performance of the issue I'm trying to debug.

```
# ./tcpretranssnoop_sdc6.d
TIME                 TCP_STATE       SRC        DST        PORT
2014 Jan  4 01:31:31  TCPS_SYN_SENT   127.0.0.1  127.0.0.1  3000
2014 Jan  4 01:31:31  TCPS_SYN_SENT   127.0.0.1  127.0.0.1  3000
2014 Jan  4 01:31:31  TCPS_SYN_SENT   127.0.0.1  127.0.0.1  3000
[…]
```

The output showed that the sessions were in the SYN_SENT state, so the packets were likely SYNs for establishing new connections. I've seen this before, when the TCP backlog queue is full due to saturation, and the kernel starts dropping SYN packets. This can be identified from "netstat -s" and the tcpListenDrop and tcpListenDropQ0 counters on SmartOS (on Linux, "SYNs to LISTEN sockets dropped" and "times the listen queue of a socket overflowed"). I was kicking myself for not checking these sooner—I should have suspected this problem for this type of benchmark.

However, these drop counters were zero. Another dead end.

Given a TCP-related issue, I looked at the remaining counters from the "netstat -s" output to study TCP more carefully, and I saw that the rate of tcpOutRsts was consistent with tcpRetransSegs. tcpOutRsts indicates TCP RST (reset) packets. Now I had a new factor to investigate: RSTs.

TCP Resets

I was curious to see packet-by-packet sequences, to see if there was a direct relationship between the RSTs and retransmits. This may also reveal other packet types that are involved. To do this, I could trace all packets or use a packet-capturing tool. I decided to try the latter to begin with, despite the higher overheads, because these tools typically do a good job of presenting packet and protocol details, which can help reveal patterns across multiple packets. I could do the same with a tracing tool, but in that case I'd need to code that presentation myself, which takes time. I tried snoop (on Linux, tcpdump) to check how the RSTs occurred, and I saw that they were happening in response to the SYNs. Why would we RST a SYN? The port was open. Was this TIME_WAIT?

Another DTrace script from my toolkit [9] showed whether packets were arriving during TIME_WAIT:

```
# ./tcptimewait.d
TIME                 TCP_STATE      SRC-IP    PORT  DST-IP    PORT FLAGS
2014 Jan  4 01:56:16 TCPS_TIME_WAIT 127.0.0.1 54170 127.0.0.1 3000  2
2014 Jan  4 01:56:16 TCPS_TIME_WAIT 127.0.0.1 50427 127.0.0.1 3000  2
2014 Jan  4 01:56:16 TCPS_TIME_WAIT 127.0.0.1 37854 127.0.0.1 3000  2
[...]
```

The output showed hundreds of packets per second. This was the TIME_WAIT issue I had thought of at the very beginning, although manifesting in a different way. Checking the ephemeral ports from the snoop output and revisiting rate counters from netstat, I could see that each of the 100 ab clients would average two successful connections per second and then block on the third. This left two connections per client in TIME_WAIT for the default of 60 seconds. So, for each second, there would be about 2 x 100 x 60 = 12,000 connections still around in TIME_WAIT, similar to the 11,000 connections I had seen earlier, which I had thought was too few to matter. Picking an ephemeral port from a range of about 33,000 when 11,000 were in use was also consistent with encountering one clash out of every three attempts. A final detail also fell into place: the "fast" rate of 5000 connections per second, seen in the earlier prstat output, lasted about two seconds. That was the time it took to reach approximately 11,000 connections in TIME_WAIT.

To find the kernel code that was causing this problem, I could follow the stack traces that led to the RSTs. I remembered that I could do this using the DTrace TCP provider I had developed while at Sun, although I couldn't remember my own syntax! A quick Internet search found my documentation, and I quickly had the stack trace responsible.

Unfortunately, the stack trace didn't look that special, with tcp_send_data() called by tcp_xmit_ctl(), which can happen for many different reasons. Fortunately, I found a gift from the kernel engineer who wrote tcp_xmit_ctl(): its first argument was a character pointer to a string explanation. Such strings are trivial to trace, and I found that it contained the text "TCPS_SYN_SENT-Bad_seq". This took me straight to the problem code, which was...familiar.

Too familiar. I started remembering more about the last time I had debugged this: we had laughed about how silly it seemed to have 60 seconds of TIME_WAIT for localhost connections and had said that that should be fixed. In fact, it had been fixed (thank you, Jerry!), but the customer had benchmarked on a system with an older illumos kernel. Linux has a different way to recycle sessions in TIME_WAIT and didn't suffer this issue in the first place. This was the reason that the competitor, on Linux, was always running five times faster, without any slowdown from retransmits.

The actual problem originates from the TCP specification: 16-bit port numbers and a lengthy TIME_WAIT. Sessions are identified by a four-tuple: client-IP:client-port:server-IP:server-port (or a three-tuple, if the server IP is not included). Because this benchmark only has one client IP, one server IP, and one server port, the only variable to uniquely identify connections is the 16-bit client ephemeral port (which by default is restricted to 32,768–65,535, so only 15-bits). After (only) thousands of connections, the chances of colliding with an old session in TIME_WAIT become great.

So the final verdict for the customer benchmark: there was a real performance issue, *and* the results were misleading. It was thought that our competitor was five times faster, but this wasn't the case for real production workloads. Node.js typically handles thousands of clients making new connections every second, not one client making thousands of new connections every second. The workaround for the benchmark was to use multiple real clients (not simulated ab clients), which brought the connection rate to around 5000 per second and steady, the same as our competitor, for this workload. The use of HTTP keep-alives was another workaround, as it avoided creating new connections altogether.

In the end, I had amazed myself: moving directly from thread time in the sleep state to TCP retransmits, by tracing which thread woke up our sleeping thread. What if I had stopped at pollsys() and not drilled down this far? That's what had happened last time I investigated: I had eventually run "netstat -s" and studied all counters, hoping for a clue, and found it. Having solved the same problem twice, using two very different approaches, gives me a rare opportunity to compare my own debugging techniques. I much prefer the direct approach that I used here—drilling down on latency and walking the wakeups.

References

[1] ab - Apache HTTP server benchmarking tool: http://httpd.apache.org/docs/2.2/programs/ab.html.

[2] illumos: http://illumos.org.

[3] USE Method: http://www.brendangregg.com/usemethod.html.

[4] SmartOS: http://smartos.org.

[5] B. Gregg, *Systems Performance: Enterprise and the Cloud* (Prentice Hall, 2013).

[6] prstat: http://illumos.org/man/1m/prstat.

[7] cv_wakeup_slow.d: https://github.com/brendangregg/dtrace-cloud-tools/blob/master/system/cv_wakeup_slow.d.

[8] tcpretranssnoop_sdc6.d: https://github.com/brendangregg/dtrace-cloud-tools/blob/master/net/tcpretranssnoop_sdc6.d.

[9] tcptimewait.d: https://github.com/brendangregg/dtrace-cloud-tools/blob/master/net/tcptimewait.d.

Large Scale Splunk Tuning

DAVID LANG

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

Splunk is a great tool for exploring your log data. It's very powerful, but still very easy to use. At lower log volumes (especially with the 500 MB/day, single-system, no-cost license) it's basically install and use. Because its paid licensing only depends on the amount of new data it sees each day, you can scale its performance and available history for "only" the cost of the systems to run it on. Very few tools can compete with Splunk for unplanned searches over very large volumes of log data. It is very practical to search hundreds of terabytes of log data and get the answer back within a minute or two.

However, as you scale up Splunk to handle these larger log volumes, you move away from the simple "anyone can do anything" install and use model and toward something that's much closer to the traditional database administration model, where you assign a small set of people to become experts in Splunk internals, watching performance, changing the configuration of Splunk, and influencing the queries sent to Splunk.

In the last article in this series [1], I talked about the ways that dashboards and reports can kill your log query system and how to change them to minimize the load they create. In this article, I will be talking about administering and configuring Splunk systems to perform the log searches. I am specifically covering the configuration of the core servers, exploring the conflicting goals that you will need to balance to match your particular data set and workload.

Overview of Splunk Internals

The main tuning configuration settings that you will want to watch are the configuration settings of your Indexes in Splunk. In Splunk "Index" is used in a way analogous to "database" in a traditional DBMS like PostgreSQL, MySQL, or Oracle. You have one instance of the main engine running, and it contains multiple databases/indexes inside it. Data is segregated between the different Indexes (although not as firmly as among different databases in a DBMS because you can issue one query that searches multiple Indexes). Unfortunately, there is no one right way to configure Splunk any more than there is a right way to configure any other DBMS. Everything is a tradeoff, where changing things will improve performance under some conditions and decrease performance in other situations. This article maps out the main configuration changes that you should be looking at, and what the benefits and costs are to provide a set of guidelines to get you started.

The one user visible configuration you will need to manage is defining what logs will end up in what Index. Users can limit their queries to specific Indexes, and user permissions can provide very solid access controls at the Index level. User permissions can provide weaker access controls within an index by automatically appending search terms to any query the user performs. I will go into more detail on the advantages and disadvantages of configuring what logs go to what Index later in the article. The other configuration changes are not user visible.

To start with, I need to go over some low-level details of how Splunk is implemented and related terminology.

Splunk is a MapReduce type of system where one or more search heads provide the search interface for the users and then parse the query, determine which Indexes need to be searched (explicitly specifying them in the search or falling back on the user's default set of Indexes), and dispatch the appropriate sub-queries to the various Indexer machines. The search head then combines the results and does any additional processing that's needed. In most use cases, the bottleneck is going to be in the data retrieval from the Indexer machines far more than the search heads.

When an Indexer receives a search from a search head, it applies timestamp filters. The data in an Index is split up into "buckets" no larger than a per-index configured size. Buckets are identified by the date range contained in that bucket, so Splunk doesn't even look in a bucket unless it contains the date range you are searching for. Once the machine determines that a bucket contains the date range, new versions of Splunk can use Bloom filters [2] to perform a second check to see whether the query term may be in that bucket. Bloom filters provide Splunk with a very small chunk of data that can be used to definitively say that the search term does NOT appear in the bucket. A Bloom filter is 128 K per bucket, and because it's so small, it's fast to search and is likely to be cached in RAM. Bloom filters cannot say that the data IS in the bucket; that requires a full index search.

Inside each bucket, Splunk keeps the raw logs in a set of gzipped files and has a large amount of index data—this time using index in its traditional database meaning of the term—data that makes it faster to search than retrieving the raw log data. This index data can easily be the majority of the storage space. I have observed that with a bucket size of 10 GB, 3–4 GB is the gzipped raw log data (holding ~30 GB of raw data), and the remaining 6–7 GB is index data. Other users have reported even worse ratios: 2 GB of compressed data (13 GB raw) with 8 GB of index data. The Splunk documentation gives a rough estimate that your raw data will compress to ~10% its original size, and Splunk will use from 10% to 110% of the raw data size for indexes. For the rest of this article, I use Index to refer to the high-level structure and index when I need to talk about this lower level data. Luckily, only Splunk admins need to deal with indexes, and even they don't need to do much with them. Unlike indexes in a traditional database, Splunk indexes require very little configuration.

Buckets go through a series of states, with size, time, count, and manual triggers to move a bucket from one state to the next.

Bucket States

HOT

- Writable: all new data that arrives is written to a Hot bucket.
- If there are multiple Hot buckets, Splunk will try to put logs with different timestamps in different buckets to minimize the range of timestamps held in each bucket. This is only effective if you have logs arriving at the same time with different timestamps.
- Roll to Warm.

WARM

- Read-only.
- May be replicated to multiple machines for resiliency.
- Must live on the same file system as Hot buckets.
- Roll to Cold.

COLD

- Same restrictions as Warm, except that they can live on a different file system.
- Roll to Frozen via an admin configurable cold2frozen script.

FROZEN

- Not searchable.
- The default cold2frozen throws away the index data from the bucket to reduce the storage requirements. Thawing such a bucket requires re-indexing the raw data.
- Not managed by Splunk. The cold2frozen script should move them outside of the directory tree that holds Cold and Thawed buckets.
- Do not need to be online. These can be moved to offline storage (tape, etc.).
- The Splunk admin runs a frozen2thawed script to copy the bucket to Thawed. If the cold2frozen script throws away the index data, the frozen2thawed script will need to re-index the raw data. This can take a noticeable amount of time.

THAWED

- Logically the equivalent of Cold.
- Must live on the same file system as Cold buckets.
- Splunk expects these buckets to appear and disappear dynamically.

By allowing the data to be stored on two different file systems, you can have a small amount of fast but expensive storage for your "hot" and "warm" buckets that contain your most recent data (which theoretically is more likely to be searched) and a much larger amount of cheaper storage to hold your older data in "cold" and "thawed" buckets.

The "read-only" status of the buckets isn't quite true. There are times when an admin should go in under the covers with some of the Splunk command-line tools and modify the buckets (e.g., to split buckets that have bad timestamp ranges or that you are splitting into two Indexes), but such work is not supported by the Splunk GUI and is frowned upon by Splunk Support unless you are doing it under the direction of the Splunk Professional Services or Support teams. If you are using Splunk data replication, the buckets will be modified to indicate which machines that bucket exists on.

To optimize your Splunk cluster, you will need to balance the following goals to match your system, log volume, and query patterns. There is no one right way to tune Splunk.

Goal #1

Make your buckets as large as you can.

The larger the bucket, the more efficient it is to search for things in the bucket. Searching through the indexes within a bucket is a O(log(n)) task, so searching through two buckets of size N will take just about twice as long as searching through a single bucket of size 2*N.

Goal #2

Appropriately set the number of hot buckets per Index.

If you have data arriving from machines that are providing significantly different timestamps, mixing this data into a single hot bucket will make each bucket's timestamp range wider by the range of timestamps considered "now" by your systems. This means that anytime a search is done against this wider time frame, this bucket will need to be searched. If you have buckets that roll every three hours, but combine local timestamps from every time zone into the same bucket, you will have to search eight times as many buckets than if you had all the systems reporting the same time.

If you have a machine whose clock is wrong and is generating logs with wildly wrong timestamps, it can be FAR worse than this (think about a system that has its clock off by months or years). If you keep logs in Splunk for long time periods, and do searches against old data routinely (security forensics searches are a good example), a problem like this can hang around for a very long time. This is one of the times where it's worth looking at manipulating the bucket contents to either split the bad data out or correct it.

If you cannot reliably set your systems to a uniform time, look at the time frame that a single bucket covers and see if you would benefit from allowing more "hot" buckets so that the different time zones each end up in their own hot bucket. If one hot bucket covers 48 hours, and you have two adjacent time zones, it's probably not worth splitting. However, if one hot bucket covers less than an hour, and you have two offices in time zones 10 hours apart, it's a very large win to have two hot buckets.

Ideally, you should use UTC instead of local time on all systems so that you also avoid daylight savings-related issues. If you have systems running on local time in different time zones, there are many ways that local times (without the time zone info) can make their way into your logs. For example, the traditional syslog timestamp does not contain time-zone information. Logs generated by applications frequently embed local times in the log messages themselves.

If you frequently have systems generate logs with wildly wrong timestamps (systems that boot with clocks off by years), add an extra hot bucket to catch these stragglers.

Goal #3

Segregate your data so that Splunk can easily rule out the need to search a bucket because it is in a different Index or because it can be eliminated via Bloom filters.

If you have logs that are distinct enough from each other that it's unlikely that you will be searching in both sets of logs at the same time (e.g., Web server and router logs), put the different logs in different Indexes. You can still search both if you need to by specifying both Indexes in your query, but if you are only searching for one type of log there will be fewer buckets to search.

Similarly, if one application generates logs that look very different, and you are likely to be searching for something that does not appear in one of the subsets of logs, putting those logs in a different Index and searching it by default will be a win because the Bloom filters will quickly exclude most of the buckets that you don't care about, so the resulting search will have far fewer buckets to search.

Goal #4

Have the logs spread out evenly over as many indexer systems as you can so that no matter what time frame you are searching for, you can utilize all of the systems. If your data is not distributed evenly across the systems, the system that has the most data will take longer to do its portion of the search and will be the limiting factor in your performance.

Splunk scales horizontally very well. As a result, the more systems that you have working on a given query, the faster your results will be returned.

With the new data replication features introduced in Splunk 5, and new management features expected in Splunk 6, you can have one box index the data and then use the Splunk replication to spread the data across multiple systems, using all of the systems to perform searches on the Index. In Splunk 5, the management tools do not allow you to do this effectively.

The other approach is to spread the data across different Indexer machines as it arrives, with each machine processing a portion of the data for that time frame. If you are using the Splunk agent to deliver the logs, you can use its load balancing mechanism to spread the data. If you are using syslog to deliver the logs, you can use any appropriate load balancing mechanism to spread the logs across the machines. Note that you do not need to try and be perfect here; even something as crude as sending several seconds' worth of logs to one machine, and then switching to the next machine for the next several seconds is going to be good enough in practice, because your search time frames are generally in minutes at the most precise and can be in years. As the load scales up, you will find that one machine cannot keep up with the data flow, and load balancing will let you use multiple machines. In this case, try to tune the bursts of traffic to each machine to be small enough that the entire burst gets handled in a reasonable time. It would not be reasonable to send all logs for a five-minute period to one machine if it takes that machine an hour to process those logs. However, it would probably be very reasonable to send all logs for a second to that machine if it only takes it 12 seconds to process the logs; it's almost certainly reasonable to send 1/100 of a second worth of logs to a machine and have them be processed in 0.12 seconds.

Remember that if the data is spread unevenly across the systems, you will end up waiting for the slowest system to finish. Think about failure conditions when you are planning this, and if you have an extended outage that causes the systems to become unbalanced, you may need to go in under the covers again to get them closer to balanced.

Goal #5

The number of indexes times the number of hot buckets per index times bucket size needs to not only fit in RAM, but leave enough RAM left over for your searches (both the state info needed for real-time searches, and the cache space needed to effectively retrieve data from disk for your searches).

Note that this goal is in direct conflict with the earlier ones. The earlier ones drive you toward more indexes, more hot buckets, and larger bucket sizes, whereas this goal is saying that you need to minimize the product of these three values.

Also, if you have hot buckets that do not receive new logs for an extended time frame, they may end up getting pushed out of cache by the OS, resulting in worse performance as they have to get pulled back in.

When you do a search, Splunk will have to do its search through the Bloom filter and indexes of that bucket. The searches through the indexes are random searches over a large volume of data. If you have very large volumes of data compared to the RAM available for disk caching on your systems (after you account for the OS, the hot buckets, and the real-time searches), you will not have this data cached and so will have to read it from disk. This is both a very common case (it's common to have a single Splunk node with multiple TBs of indexed data, and it's unusual to have more than a few tens of GB of RAM available for caching) and a worst case random read load for spinning drives. If the version of Splunk you are running allows it, and you can afford it, putting the Bloom filter files on SSD storage can go a long way towards mitigating the cost of finding and reading them in after they are pushed out of the systems' disk cache.

Because a new log arriving can result in changes to the bucket indexes that result in rewriting large portions of the index data, you *really* want to have all of your hot buckets in RAM at all times; if any of them get pushed out to disk, you are likely to have to retrieve the data and rewrite it in the near future.

Remember that you do not have to put logs for every Index on every machine; you can split the Splunk cluster into multiple sub-clusters, with each sub-cluster holding specific types of logs.

Goal #6

Minimize the number of buckets that need to be retrieved for each query.

This goal requires a lot more explanation because it is in direct conflict with the earlier goals.

If you commonly do searches across different Indexes (if you have one Index per application, but need to look for someone accessing any application, for example), consider combining the Indexes together. The resulting Index will be less efficient to search than either Index on its own, but it may be significantly faster to search the combined Index than the two separate Indexes for a single query.

In addition, because the bucket sizes are constant, you may find that the time frames you commonly search are poorly matched with the time ranges that the buckets end up containing.

For example, given four log sources, each generating the same amount of data, you put them each into a separate Index, and the bucket size combined with your data rate means that each bucket contains four hours' worth of logs. A search for something across all four log sources over the last hour will still have to search one bucket per Index for a total of four buckets.

If, however, you were to combine all four log sources into a single Index, then each bucket will contain one hour's worth of logs, and a search over the last hour will only have to search a total of one bucket.

On the other hand, if your typical search is instead limited to a single log source, but it covers the last day, putting each application in its own Index will require searching six buckets, while having all four applications in the same Index will require searching 24 buckets.

Unless you are really sure that you have people doing a lot of searches of a specific set of data, I suggest starting out with everything in one Index and splitting it out later as you discover the true query patterns of your users.

Even if you have people interested in only a specific set of data, if they don't bother restricting their search to a particular Index, their query will be evaluated against all the indexes that their user permissions default to.

Special Case: Summary Log Data

Summary log data is a special case. Because it's fairly small and, as a result, queries against a time-range of summary data tends to want to retrieve all the data, it can be a good idea to have a separate Index for your summary data. Instead of setting the bucket size for the summary data Index, set the maxHotSpan-Secs parameter to either 86400 (1 day) or 3600 (1 hour), and have Splunk rotate the hot buckets every hour every day at midnight or on the hour. The buckets are smaller, so a report over a massive time range will be a little less efficient than a large bucket, but the smaller bucket sizes match the time limits of your queries nicely, and it's much better to search for an hour's worth of data in a one-day bucket than in a three-week bucket. Summary data is also going to be searched far more frequently than normal data (largely due to dashboards).

As a result of this, and the fact that the data tends to be small, in many cases it makes sense to set up two separate sets of servers: one to handle your summary data, the other to handle your main data. The servers you use to handle the summary data do not need to have the super-beefy disk systems that your main data systems have; it's likely that you can set up systems that will keep the working set of the data that your users are querying in RAM. At that point, CPU will become your limiting factor instead of the disk I/O that is normally the first limitation. By setting up a separate set of systems to serve the summary data to dashboard users, you ensure that the dashboards are not going to impact the performance of your main systems: the worst they will do is slow each other down as well as slow report generation.

Conclusion

Splunk can be a wonderful tool for exploring your logs. It works very well for a wide range of user expertise. But if you don't have someone with a high level of expertise managing the system (very similar to the way that you would not dream of running a large Oracle system without a good DBA), you are likely to end up with a very poorly performing system. I've seen a Splunk cluster configured such that it would have required 40 TBs of RAM to cache all the hot buckets Splunk was trying to manage. I've also seen Splunk get to the point where the dashboards were showing data over three hours old because of poor tuning and bad usage patterns.

I am not saying that you need to prevent people from using Splunk, even for dashboards and reports. However, you do need to avoid giving everyone admin rights to Splunk, and you need to have a team of experts monitor and configure Splunk and how it's used so that you can maintain good performance of your system. If you don't do this, you are going to either end up building a cluster that is huge and expensive compared to what it really needs to be, or have it slow to a crawl.

This article is the latest in this series on logging topics [3]. If you have requests for topics for future articles, please email me (david@lang.hm) or Rik Farrow (rik@usenix.org) with your suggestions.

References

[1] "Logging Reports and Dashboards," *;login:*, vol. 39, no. 1, February 2014: https://www.usenix.org/publications/login/feb14/logging-reports-dashboards.

[2] http://docs.splunk.com/Documentation/Splunk/5.0.2/indexer/Bloomfilters.

[3] http://www.usenix.org/login/david-lang-series.

/var/log/manager

Let's Find Someone to Blame

ANDY SEELY

Andy Seely is the manager of an IT engineering division, customer-site Chief Engineer, and a computer science instructor for the University of Maryland University College. His wife Heather is his init process and his sons Marek and Ivo are always on the run queue.
andy@yankeetown.com

If two people are in a boat and lost at sea, both have to row to survive; one doesn't get to be the captain and the other the sailor. In a large organization with layers of management and silos of responsibility, figuring out individual responsibilities and root cause of a failure is a lot more difficult than just blaming someone and moving on. Assigning blame is actually pretty easy. Truly understanding failure and finding a way to appease the top while improving the organization's overall effectiveness takes real management skill.

The Manager's Problem: The Product Release Failed

The VP of communications and technology came to me after we got news about a problem with our latest product release. "I want you to deal with your engineer. He totally messed this up." That's my job. The engineer works for me. I'm responsible for his actions. The engineer didn't do anything wrong, but the VP needed blood from his own organization and had already decided who he was going to blame: my engineer.

I'm a senior manager in the organization, but it's not like I own the company. At the executive level above me, vice presidents like mine can have motivations that are sometimes mysterious. My goals are simple: empower people to do their best work towards making the systems perform business functions correctly and within performance parameters, as cheaply as possible. Sometimes that means knowing people, understanding their motivations, and clearing a path for them. And sometimes that means taking a bullet. This day, I took the bullet.

The Manager's Choice: Assigning Blame or Understanding the Bigger Picture

It's a complex organization, with multiple echelons of the organization and several management chains involved. At least six people are involved in the product, not including their individual managers and chains of command. It's a complex product. From requirement, to build, test, security, QA, release, to deployment, through the occasional post-production error, the product passes through a lot of gates and a lot of hands. The process is mature and usually works well, but when something goes awry it's difficult to find clear fault. In my opinion, quick blame is really a luxury anyway; understanding the true root cause of a failure is always about learning and improving and, honestly, sometimes about blame.

My investigation showed that our release was done correctly, by the book, with no problems. Maybe it was a little rushed because of a shorter than usual operations deadline. Maybe the engineer who puts the package together had this "normal rush job" to do and he had another rush job to do at the same time. Maybe the security review was done by the second-string technician this time. Maybe there were externalities for which we didn't test, because our test environment is not set up for everything under the sun. Maybe a lot of things happened, but ultimately the release passed all the gates and met the deadline.

We only got word of a problem after the release was in production for a few days. A user of a remote service provided by an external company was getting failures that were traced back to a local Java dependency. We had updated our Java Runtime Environment to the current version recommended by Oracle and all our internal Java dependencies passed, but this remote application had some poorly coded dependency on the minor revision number. This user had a direct line to the top of our organization and made his problem a high-level issue that rolled downhill onto my team.

The Manager's Challenge: Threading the Least-Worst Path Through Failure

I'm the manager of the division. I work for a VP and I have branch managers who work for me. I am dealing with a product release that is considered a failure by those above me and a success by those below me.

There are three ways this might play out.

1. I defend my team to my VP. They did it right, the release was textbook-correct, and the failure case is not on them, it's on the external organization that didn't update their code to work with ours. The result will be that the VP will have to show his bosses he took action, and since he can't control the external organization he'll look at me as the thing he can control. After all, if the release was both "textbook" and "failure," then the textbook, my textbook, was flawed and I refused to take action. I sacrifice either my credibility or my job, and neither result will actually help improve future releases.
2. I act as pass-through for the heat. I write up the integration branch manager, and then he writes up his engineers who built the release. Or I just save him the trouble and write him up and fire his guy for him. Or I fire him and his guy. Make an example out of everyone and prove myself to be "he who manages by fear," creating a demoralized workforce. This path is easy. Holding others accountable rarely costs you your own skin, as long as you're willing to blame employees for causing their own demoralization. This approach will result in losing the people whom you have depended upon the most, who know the systems and processes the best, and it will set the stage for a working environment where the only people who stay are those for whom fear is actually an effective motivation.
3. I find the middle ground where my VP can get satisfaction, my crew can be proud of their work, and people can get the chance to improve the process and release failure-resistant products. As a side effect, I can show organizational maturity through flexibility and introspection. I stand in front of the VP and take the blame, which is not entirely misplaced because it's my organization's release that was found wanting, but I defy the demand for counseling or firing people. I turn to my team and hold them accountable, without blaming them. I challenge them to find a way to prevent this type of failure in the future. Not just this specific failure, but to define this as a class of failure and fix the process. Evolve.

Empower and Challenge People to Get Results that Really Matter

If done well, the middle-ground approach sets the stage for future success and helps to mature the organization. The team sees that the manager took responsibility and didn't just pass blame directly through. A team that already takes pride in their product will respond well to a challenge to make that product stronger.

I'm the manager. I take the heat and hold the line, while giving the team a new goal. The team sees how much I have on the line, and they'll work hard for me because I put myself out there for them, and they'll produce an improved product. My VP gets what he needs to answer his own masters, and maybe is slower to seek blood the next time. And me? I get the pleasure of having that rare opportunity not to be the boss, not to be a manager, but to be a leader, equal parts showing the way and being fully invested in the outcome of the whole team. As a leader, if my own skin isn't in the game then nothing I do really counts.

How Did It End?

We found the actual culprit for the failed release. As everyone had said, it was a textbook product release. We discovered that a chapter of our textbook was, indeed, flawed. Every application had passed its respective application validation checklist, but there was no governance for how those checklists were reviewed and updated. We had been relying on sysadmins and engineers to create the validation checklists, and in some cases they had no idea how the applications were actually used by functional operators.

A simple solution was to add a user validation section to every application validation checklist in the product release package; before the product is released, key users of each application will be asked to validate the function of their applications. This added step gave us some immediate benefits. We gained improved user engagement, which gave us greater understanding of how our product was used, and every release was now checked by the people who actually depend on it. Now if a user discovers a post-release product failure in the future, he's vastly more likely to call his new friends in the engineering shop than to call the CEO. Through understanding real responsibility and holding the line on the blame game, we found the right path to keep our best people on the team and deliver a better product. I'm the manager. That's my job.

23rd USENIX Security Symposium

AUGUST 20–22, 2014 • SAN DIEGO, CA

The USENIX Security Symposium brings together researchers, practitioners, system administrators, system programmers, and others interested in the latest advances in the security of computer systems and networks. The Symposium will include a 3-day technical program with refereed papers, invited talks, posters, panel discussions, and Birds-of-a-Feather sessions.

The following co-located events will precede the Symposium.

EVT/WOTE '14: 2014 Electronic Voting Technology Workshop/Workshop on Trustworthy Elections
August 18–19, 2014
www.usenix.org/evtwote14

USENIX Journal of Election Technology and Systems (JETS)
Published in conjunction with EVT/WOTE
www.usenix.org/jets

CSET '14: 7th Workshop on Cyber Security Experimentation and Test
August 18, 2014
www.usenix.org/cset14
Submissions due: April 25, 2014

NEW! 3GSE '14: 2014 USENIX Summit on Gaming, Games, and Gamification in Security Education
August 18, 2014
www.usenix.org/3gse14
Invited submissions due: May 6, 2014

FOCI '14: 4th USENIX Workshop on Free and Open Communications on the Internet
August 18, 2014
www.usenix.org/foci14
Submissions due: May 13, 2014

HotSec '14: 2014 USENIX Summit on Hot Topics in Security
August 19, 2014
www.usenix.org/hotsec14

HealthTech '14: 2014 USENIX Summit on Health Information Technologies
Safety, Security, Privacy, and Interoperability of Health Information Technologies
August 19, 2014
www.usenix.org/healthtech14
Submissions due: May 9, 2014

WOOT '14: 8th USENIX Workshop on Offensive Technologies
August 19, 2014

usenix THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

www.usenix.org/sec14

Stay Connected...

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

SDN Is DevOps for Networking

ROB SHERWOOD

Rob serves as the CTO for Big Switch Networks, where he spends his time internally leading software architecture and externally evangelizing SDN to customers and partners. Rob is an active contributor to open source projects such as Switch Light and Floodlight as well as the Open Compute Project. He was the former chair of the ONF's Architecture and Framework Working Group as well as vice-chair of the ONF's Testing and Interoperability Working Group. Rob prototyped the first OpenFlow-based network hypervisor, the FlowVisor, allowing production and experimental traffic to safely co-exist on the same physical network and is involved in various standards efforts and partner and customer engagements. Rob holds a PhD in computer science from the University of Maryland, College Park.
rob.sherwood@bigswitch.com

Caught in a perfect storm of technology trends—including public and private cloud, Bring-Your-Own-Device (BYOD), converged storage, and VoIP—computer network management is reaching unprecedented levels of complexity. However, unlike server administrators whose tools have evolved with the times, network administrators are stuck using 20+-year-old box-by-box management tools. A new technology trend, Software-Defined Networking (SDN), promises to simplify network management. Although it seems like every vendor has its own definition of SDN, in this article, I make the claim that SDN is to networking what the DevOps movement is to server management: a way of making systems management easier to manage by adding programmable APIs that enable better automation, centralization, and debugging. In this article, I try to provide background on SDN, to snapshot its current and highly fluid state, and end with some predictions for what to expect next.

Networking Needs a "DevOps"

All types of networks, including campus, datacenter, branch office, wide area, and access networks, are growing at unprecedented speeds. More people with more devices are coming online and are accessing an increasing plethora of data. Although this is good for society at large, the reality is that networks themselves are becoming increasingly difficult for "mere mortals" to manage. In the past, sensitive data would exist on a single, dedicated, physical server in a fixed location with a clear physical DMZ policy "choke point" as the divide between the trusted and untrusted parts of the network. Today, sensitive data can be spread across multiple databases potentially distributed throughout the cloud on virtual machines that change physical location depending on load; thus, the single policy "choke-point" is a thing of the past.

While low-level packet forwarding devices have made amazing advances with speeds moving from 100 Mb/s server ports to 10 Gb/s and beyond, the management tools needed to operate and debug these devices have stagnated. Indeed, operators of today use the same basic command-line syntax and tools to configure routers as when I first started administering networks 20 years ago. The only thing that seems to have changed is that we used to telnet to these boxes but now we use SSH! Furthermore, network administrators are caught between a rock and a hard place because the management interfaces for the network devices are typically closed, vertically integrated systems that resist enhancement or replacement.

Despite going through the same growing pains, server administrators dodged these problems with a variety of automation and centralization tools that can roughly be grouped under the term "DevOps." DevOps infuses traditional server administration with best practices from software engineering, including abstraction, automation, centralization, release management, and testing. The server ecosystem is quite different from networking because it has many open interfaces: server admins can supplement, enhance, or replace software components of their systems, including configuration files, whole applications, device drivers, libraries, or even the entire operating system if desired. As a result, server administrators were able to manage growing server complexity by replacing and automating critical components of their management stack with tools such as Puppet [1], Chef [2], and others. In other words, although server administrators have the same problems in terms of scale and complexity as network administrators, they were able to solve their problem with DevOps-style deployments because the server ecosystem has open and programmable interfaces.

SDN Promises an Interface to Unlock Networking

Centralization, automation, and better debugging sound like good goals, but the closed and vertically integrated nature of most switches and routers makes it unclear how to apply them to networking. SDN promises to create an application programming interface (API) for networking and thus unlock DevOps' same desirable properties of automation, centralization, and testing.

The term SDN was first coined in an *MIT Technology Review* article [3] by comparing the shift in networking to the shift in radio technology with the advance from software defined radios. However, the term is perhaps misleading because almost all networking devices contain a mix of hardware and software components. This ambiguity was leveraged and exacerbated by a litany of companies trying to re-brand products under the "SDN" umbrella and effectively join the SDN bandwagon. As a result, much of the technical merit of SDN has been lost in the noise.

The SDN movement, originating from Stanford University circa 2007, was first exemplified by the OpenFlow protocol. OpenFlow is an open protocol and is currently maintained by the vendor-neutral Open Networking Foundation [4]. OpenFlow exposes a remote API for managing the low-level packet forwarding portions of network devices, including switches, routers, access points, and the like. At a high level, OpenFlow abstracts packet forwarding devices as a series of "match action" tables. That is, when a packet arrives at a device, it is processed through a series of prioritized lookup tables of the form "if the packet matches *MATCH,* then apply *LIST OF ACTIONS,"* where the list of actions can be anything from "send packet out port X," "decrement TTL," or rewrite one of the packet header fields. By creating this abstraction layer and interface, network admins can, in a programmable way, manage the forwarding rules of their networks in an automated and centralized manner.

Many-Layered APIs of SDN

From the first successes of OpenFlow [5], SDN began to expand and consider new use cases and deployments. As with any vibrant software ecosystem, many APIs—both complementary and competing—have begun to emerge. In addition to APIs like OpenFlow for managing packet forwarding logic, interfaces for managing configuration, tunneling, as well as more traditional

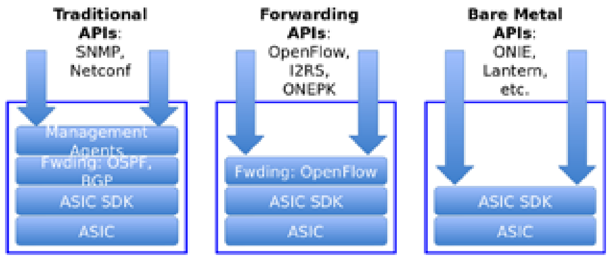

Traditional APIs: SNMP, Netconf
Management Agents
Fwding: OSPF, BGP
ASIC SDK
ASIC
Forwarding APIs: OpenFlow, I2RS, ONEPK
Fwding: OpenFlow
ASIC SDK
ASIC
Bare Metal APIs: ONIE, Lantern, etc.
ASIC SDK
ASIC

Figure 1: SDN has many layers, from the traditional APIs, to the forwarding APIs that were the first target of SDN, to the bare metal APIs.

APIs for statistics monitoring and debugging are being viewed as SDN. Most recently, much like with servers, the lowest-level "bare metal" hardware APIs are being exposed, allowing enterprising startup companies and DIY types to write their own network operating systems from the ground up. In other words, SDN is bringing the networking ecosystem closer to the server DevOps ecosystem where an administrator can choose the right API/tool for the task and automate and centralize common tasks.

As with any complex and rapidly evolving system, tracking all of the APIs, protocols, ideas, and works-in-progress is impractical, but here I try to provide a hopefully representative snapshot of the state of SDN. Figure 1 provides a visible map of some of the existing layers.

Forwarding Plane APIs

Probably the most important and novel aspect of SDN is the ability to programmatically manage low-level packet forwarding. OpenFlow itself has evolved quite a bit since its debut 1.0 release in 2009. More modern versions of OpenFlow have added support for richer packet actions (e.g., NAT, tunneling, metering), more extensible matches, multiple tables, IPv6 support, and even batched "bundled" commands with the most recent version: OpenFlow 1.4.0 [6]. In addition to new forwarding capabilities, the Open Networking Foundation (ONF) is exploring better abstractions for wired forwarding hardware as well as for optical and wireless technologies. Rather than replacing traditional routing forwarding decisions, the IETF's Interface to the Routing System (I2RS [7]) seeks to provide an API for merging programmatic packet forwarding with packet forwarding decisions inferred from traditional routing protocols like BGP, IS-IS, OSPF, and others. Not to be left out, traditional networking vendors have created their own forwarding APIs, including Cisco's onePK and Juniper's Junos Space.

Configuration and Statistics APIs

Besides low-level packet forwarding, networking devices have a dizzying array of tunable configuration parameters and statistics. Many protocols, such as SNMP and Netconf that long predated OpenFlow exposed APIs ("MIBs" in SNMP, "schemas" in Netconf), allow network admins to tweak configuration settings and monitor statistics like port counters. Newer APIs, like ONF's OpenFlow Config protocol [8] and Open vSwitch's DB management API [9], supplement existing APIs by adding support for managing tunnels and virtual switches (i.e., by adding and removing virtual ports). Additionally, many of these APIs have support to enable and configure packet sampling protocols like NetFlow and sFlow, which are critical for in-depth traffic analysis.

Bare Metal and Open Hardware APIs

Whereas the above APIs build on top of existing vendor software, it is increasingly possible to write directly to the low-level hardware APIs and replace vendor software altogether. By comparison, if writing packet forwarding rules is like writing your own application, then writing to the bare metal hardware is like writing your own operating system. Although writing the entire network stack is not for the faint-of-heart, it can be necessary to overcome limitations of existing vendor stacks or to accomplish something completely revolutionary. Writing to the bare metal is made possible by two recent changes in the ecosystem: a standardized network device boot loader and open ASIC APIs.

The Open Network Install Environment (ONIE [10]) is an open source boot loader available for an increasing number of networking devices, particularly datacenter switches. In server terms, ONIE provides functionality that is one part PC BIOS and one part grub/lilo/sysimage. A network admin can use ONIE to add/remove/reset the switch operating system over the network. In other words, using ONIE, it is possible to network boot (or even dual boot!) an arbitrary network operating system on to an ONIE-enabled network device. Think of it as PXE for switches and routers. ONIE is hosted and sponsored by the Open Compute Project (OCP [11]), which, among other aspects, includes open hardware designs and specifications for networking devices.

To achieve high speeds, modern networking typically requires special purpose hardware, such as an Application-Specific Integrated Circuit (ASIC). Historically, the APIs to program these ASICs have been closed and access to them tightly controlled via strict non-disclosure agreements. However, more recently, ASIC manufacturers are moving to the "bare metal" bandwagon and have started to make the APIs public. For example, ASIC manufacturers Centec and Mellanox have begun to publish their APIs in their Lantern [12] and Open Ethernet [13] projects, respectively. Other ASIC manufacturers seem likely to follow suit, so this trend seems likely to increase over time.

Impact from Market Forces

Although SDN is primarily a technology movement, it would be an error to assume that its traction is purely a result of a superior architecture. As technologists, we like to ignore the economics, but history is filled with technologies that didn't succeed despite superior design. In particular, technologies similar to SDN have come and gone in the past without comparable traction, including IETF's ForCES [14] and the field of active networking. So a critical question is, why is SDN achieving industry traction where similar technologies have not?

The answer is that the underlying market forces of networking have significantly changed. Large datacenters mean that more money is being spent on networking than ever before, which encourages both more competition as well as bigger gains from commodities of scale. Historically, packet forwarding ASICs were only created by pure networking vendors for inclusion into their own vertically integrated products. As a result, the market rewarded vertically integrated closed systems because that best protected the companies' ASIC investments. But, with the rise of large datacenters, sufficient ASICs were being sold that highly specialized "ASIC only" companies became commercially viable. Soon, companies like Broadcom, Marvell, Fulcrum, and others began to create switching ASICs and sell them to others without owning the full solution. Competition for this new commodity "merchant" silicon space increased, and now we are beginning to see strong market forces come into play in terms of lower costs and additional features. This is all because merchant silicon companies have the incentive to sell more and better ASICs—not more and better boxes. It is a result of this competition that these same companies are breaking industry norms and publishing their APIs—and thus enabling SDN.

Another effect of large datacenters is the convergence of compute, storage, and networking. Administrators are increasingly buying these resources as integrated solutions, and vendors are reacting in turn. The result is that traditional networking companies are starting to sell products that integrate compute and storage (e.g., Cisco's UCS product), and traditional computer companies are starting to acquire networking companies (e.g., HP bought 3Com, Dell bought Force10, IBM bought Blade Networks). The once very stable networking market is full of new and significant competition, with each company looking for ways to differentiate itself from the rest—including opening up networking devices by implementing SDN protocols like OpenFlow.

Conclusions and Predictions

Networking administrators are adopting SDN for many of the same reasons that server administrators adopted DevOps: automation, centralization, and ease of debugging. Historically, network devices have been vertically integrated closed software stacks with few mechanisms to replace or extend their functionality. However, recent changes in the market are causing vendors to shift their business models and open up their devices to programmable access through a suite of APIs. The result appears to be a trend towards a more extensible and vibrant third-party software-driven networking ecosystem.

Perhaps more interesting than any one specific API are the applications that are enabled by using combinations of APIs. For example, imagine an application that makes API calls to all of the devices in the network to set up sFlow sessions, monitor the dynamically changing traffic, and then make further API calls to readjust traffic engineering policies via OpenFlow. Such combinations will allow networks to be more easily managed and scale up to the demands from BYOD, VoIP, and future technology trends.

In terms of predictions, my big claim is this: after 20+ years of closed software stacks in networking devices, the genie is out of the bottle. I believe that as in the transition from the IBM mainframe to the PC or from closed cell phones to modern, open API smartphones, we will see networking go through a renaissance. We will see switch operating systems and applications that are entirely open source, and applications that do more niche and specialized tasks. We will see the cost of hardware drop significantly: just to put a number to it, I believe we will see the cost of 10G Ethernet switches drop below $75 per port before 2015. This explosion of new ideas, lower cost hardware, and innovative networking features will change how networking consumers view their networks. In other words, I believe that with an open network, operators will be empowered to create and deploy innovative new features that will change networking from a cost center into a new source of revenue in terms of novel products for their customers. The really fun question becomes, what will be the killer app that no one thought of until everyone needed it?

References

[1] Puppet: http://www.puppetlabs.com.

[2] Chef: http://www.getchef.com.

[3] Kate Greene, "10 Breakthrough Technologies: TR10—Software-Defined Networking," *MIT Technology Review*, March/April 2009: http://www2.technologyreview.com/article/412194/tr10-software-defined-networking/.

[4] Open Networking Foundation: https://www.opennetworking.org.

[5] Open Networking Foundation, Solution Briefs: https://www.opennetworking.org/sdn-resources/sdn-library/solution-briefs.

[6] Open Networking Foundation, OpenFlow 1.4.0 Wire Protocol Specification: https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.4.0.pdf.

[7] I2RS: https://datatracker.ietf.org/wg/i2rs/charter/.

[8] OpenFlow Config Protocol: https://www.opennetworking.org/sdn-resources/onf-specifications/openflow-config.

[9] Open vSwitch's DB management API: http://www.openvswitch.org.

[10] Open Network Install Environment (ONIE): http://onie.github.io/onie/docs/overview/index.html.

[11] Open Compute Project: Networking: http://www.opencompute.org/projects/networking/.

[12] Centec Lantern ASIC APIs: http://www.centecnetworks.com/en/OpenSourceList.asp?ID=260.

[13] Mellanox Open Ethernet Project: http://www.mellanox.com/openethernet/.

[14] IETF's ForCES: http://datatracker.ietf.org/wg/forces/charter/.

Musings and Hacks on DHCP

DOUG HUGHES

Doug Hughes is the manager for the infrastructure team at D. E. Shaw Research, LLC. in Manhattan. He is a past LOPSA board member and was the LISA 2011 conference co-chair. Doug fell into system administration accidently after acquiring a B.E. in computer engineering, and decided that it suited him. doug@will.to

When you think of DHCP, where do your thoughts tend? Perhaps to assigning temporary addresses to devices. Perhaps to lease maintenance and netblock assignment. Perhaps as the original extension to BOOTP [1]. DHCP is a lot of things to a lot of people, sometimes maligned, often underappreciated, and quite often used without full comprehension. After an introduction, I plan to point out some interesting possibilities, implementation issues, and potential novel uses for DHCP that most sites likely haven't considered. At the end, I'll address our implementation and how it increases our operational efficiency.

Introduction

The Dynamic Host Configuration Protocol (DHCP) [2] was defined in October of 1993 as an extension to the Bootstrap Protocol (BOOTP) then prevalent. Since those primordial days, it has undergone many revisions, extensions, and has become de rigueur at most sites. The original raison d'être of DHCP was to enable one to dynamically assign addresses to a pool of machines. This was a boon for ISPs and corporations that had mobile machines or a limited set of modems where users would dial in, set up a PPP or SL/IP connection, and be assigned an IP address. Enter DHCP, a way to give a lease on an address to a connection that could be reused by somebody else later. But DHCP has many more extensions over BOOTP as well. I'll add some color to these shortly.

Later, in 1999, DHCP got a new lease on life (so to speak) with the invention of Preboot eXecution Environment (PXE) [3] by Intel and SystemSoft. PXE was, and remains, a marvelous invention allowing for embedding DHCP (and TFTP, or Trivial File Transfer Protocol) into the NIC so that it can download some small bit of code to enable bootstrapping and installing machines. The combination of DHCP with TFTP [4], and some identifiers like UUID and GUID with an API, was a master stroke for the sites that needed a way to install machines with as little interaction as possible. To this day, TFTP is still used even though it was originally implemented in 1981! These days, though, TFTP is often used as a means to load a small executable like PXELINUX [5], which then will do some further loading via HTTP or NFS or another mechanism. Even so, TFTP still has widespread use in the network provider space. But this article isn't about either PXE or TFTP.

The Workings of DHCP

DHCP has a small number of things that are mandatory to send to the host and come right at the top of the request and reply packets. Many are carried forward from BOOTP to retain backward compatibility, with some minor changes to extend the capabilities. The important ones are outlined in Table 1.

Let's briefly refocus on the "Options." Many of these are defined in their own RFCs. Some are vendor-specific tags that extend the protocol in particular ways, like defining specific addresses of servers for particular protocols. Some are functionally obsolete, like "NDS Server" or "Impress Server." I will have more to say about options later.

It's important to note that DHCP protocol has three modes for assigning addresses: automatic allocation, dynamic allocation, and manual allocation. Automatic allocation assigns a

Name	Description
Operation	Request or reply (client or server)
Transaction ID	Used for discriminating among multiple clients for leases
Lease time	Number of seconds before lease expires
Ciaddr	Client IP address (in client request; used for bound, renewing, and rebinding clients)
Yiaddr	Your (client) IP address—given by server in reply
Siaddr	Address to use for next server, typically the TFTP server for fetching code
Giaddr	Gateway address; when a proxy is involved
Chaddr	Client hardware address (e.g., Ethernet MAC)
Hostname	Up to 64 characters, null terminated
Boot file	128-character path for boot file on server
Options	A set of tags and values

Table 1: DHCP and BOOTP essential tags

permanent IP address to a client from an address pool. Dynamic allocation assigns an address to a client from a pool for a limited period of time, which may be renewed. Manual allocation allows a network or system administrator to map the MAC address to a specific IP address. Dynamic allocation is the mode that allows automatic reuse of IP addresses for transient clients.

On DHCP Options

Every DHCP option consists of two bytes. The first byte is the option identifier, or tag. The second byte is the length of the option data in bytes. So you can see that there are built-in limits on options. There can be only 256 options and every option can be only up to 255 bytes long. A length of 0 is valid for Boolean type flags, and option tags 0 and 255 are special. Thus, since an IPv4 address is four bytes, every option tag that references an IPv4 address (TFTP server, DNS server, etc.) is four bytes long (plus the two preceding bytes with the tag number and the data length).

Other interesting tidbits:

- Some string options are required to be NULL terminated. Others are not. (The null is included in the length!)
- As mentioned, Option 255 is special and has a 0 byte length. It is added by the server to signal the end of the DHCP reply.
- Option 0 is also special and is a pad as defined in RFC 2132. It is used to cause subsequent fields to align on word boundaries if necessary.
- Option 55 carries a parameter request list. It is inserted by the client and includes the tag numbers of options that the client wishes to receive. The byte length field of Option 55 holds the count of the number of tags, and each byte after the length is an option tag number. The DHCP server isn't required to answer every tag if it doesn't have information for that tag, but it must try to provide any tags that it does answer in the order requested by the client. Thus, if the client requests options 5, 120, 40, 35, 39, 20, then the server, when composing its reply, should insert them in the reply packet in that same order.
- Wireshark will decode options for you. It is quite illuminating.
- Option 82 is quite interesting, and figures heavily into our implementation. Stay tuned.

What Do We Need?

With the obligatory introductory technical information out of the way, let's take a step back to look at business objectives. At my organization, we have a number of custom supercomputer machines for molecular chemistry research. Each machine is functionally identical, with a number of ASIC boards, a number of commodity computers connected to the boards with a custom PCI card, and a number of commodity network switches connecting the off-the-shelf computers into the network for storage and supporting software.

When a commodity node fails, we want to replace it quickly. The new node must have the same name, same location, and same IP address as the old one, because it will be doing the same thing. Also, if a switch fails, we want to be able to put a new one in place, plug it in, and have it auto-configure itself, including the correct VLANs, switch port labels, and sundries. It is impractical to have a spare sitting around preconfigured for every possibility of failed device. It is very desirable to have a factory-default box ready to plug in, install itself, and be ready to go in a couple of minutes. DHCP can help manage this. It was on the search to solve these issues that we discovered Option 82.

Additionally, my organization tends to buy servers in units of a rack, to save time, labor, and money on partial integration. This allows us to take advantage of third-party integration services for getting all of the machines cabled, labeled, IPMI addresses set, tested, and ready to deliver; integrators pre-stress all manufacturer-delivered machines for early failures. Once all of the preliminary work is done in this fashion, we can receive the entire bundle in a crated rack, roll it into place, plug in the power distribution unit and the network uplink cables, and install all of the servers at one time. Because all of the IPMI addresses are set, all we need to do is configure the switch(es) to the appropriate VLAN(s), connect the power and network uplinks, and start installing all the machines remotely.

One last component of labor and delivery speed that caused us to expend a considerable amount of time and effort was the fact that, to install the nodes, we needed to assign IP addresses. Referring back to the DHCP "modes," DHCP Dynamic mode is unfavorable because we want all of the nodes to have constant DNS resolution. We could use dynamic DNS for this, but adding that infrastructure and having IPs and names in a predefined order to facilitate subsequent debugging has positive benefits and fewer moving parts. Any statistics or history that is indexed by an IP address when using dynamic mode would be lost on a random DHCP reassignment later on.

Up to this point, we've had our integrator supply us with a list of all of the MAC addresses of every node in a rack integration spreadsheet. This spreadsheet has the MAC address of each eth0, eth1, and IPMI for each server, its rack position, name, serial number (for later RMA), Ethernet switch port designation, switch name (if more than 40 nodes in a rack), and PDU receptacle (if switched PDU). The MAC address collection in particular adds a lot to delivery time because it requires the integrator to gather all of the extra MAC data and carefully collate it; serial numbers are generally more accessible and do not require booting the machine. It also means we need to add all of the MAC addresses into isc-dhcpd. In our case, that means populating them into a database with the host record, so it's not that hard, but it is tedious and time-consuming and leads to slower delivery and installation.

The most common DHCP server in use today is the Internet Systems Consortium's dhcpd [7] (isc-dhcpd). It has a number of interesting features: for instance, client groups and subgroups, support for dynamic DNS updates, and access lists. Until now, we have been vigorous users of isc-dhcpd for our commodity node installations. Then our eyes were opened with the previously undiscovered utility of…

Option 82

DHCP Option 82 is defined in RFC 3046 and is a bit of a lesser known gem. It was devised, in part, as a solution for cable modem, DSL, and other providers with a high port count that want to assign IP addresses to a particular subscriber line statically without having to worry about exceeding dynamic capacity of a pool. It is still heavily in use today by that same contingent, but with some growing use outside of that. The rest of this article discusses our attempts to use and our final application of Option 82. But, first, I'll go over a few technical details.

Option 82 is called the Relay Agent Information Option. A relay agent is any device that listens for DHCP requests on one subnet and forwards them to a DHCP server on a remote network. DHCP is inherently a Layer 2 protocol and uses link-level broadcast technology to find a server. Relay agents enable DHCP to be Layer 3 capable. Relay agents are in common use at sites where there are many VLANs (or subnets) because it prevents the need for having a DHCP server per VLAN. Relay agents (typically managed network routers) are, among other things, required to add a gateway address into the request so that the server can send the reply back via the relay agent.

One uncommon thing about Option 82 is that it includes two sub-options. The length byte of Option 82 contains the combined length of the two sub-options and their sub-tags. Because there are currently only two sub-options (they automatically have room for up to 256), these are tagged 1 and 2. Sub-option 1 is the Agent Circuit ID and sub-option 2 is the Remote ID. Each of the two sub-options is voluntary. A relay agent may insert one or both into a DHCP request or skip it entirely. Additionally, the encoding of a sub-option is left up to the implementer. A DHCP server must include the entire option in its reply, verbatim.

- Agent Circuit ID: This is intended to be used as a port identifier for the agent upon which the request was received. It is left up to the vendor to determine how to encode the port (e.g., ASCII or integer, numeric, or alpha-numeric). Some vendors include VLAN and blade information for chassis switches along with the port, while others include only a port name or number. Cable modems often include the virtual circuit of the subscriber.
- Remote ID: If included, this signifies the device acting as the agent. Again, it is not specified in the RFC how this should be encoded, so it could be an ASCII switch name, a hexadecimal encoding of a serial number or VLAN IP address where the request is received, or a caller-ID for a dial-up connection. Because the RFC requires the remote ID to be globally unique, many vendors use an encoded or literal serial number but allow the administrator to override this with an arbitrary name. We use the unique name of the switch where possible.

In big networks, it is not uncommon to have multiple levels of relay agent between the source network and a destination server. Different switch vendors handle this in different ways. Some allow you to append or overwrite the remote ID and circuit ID information with new information, at the administrator's choice. Some will only overwrite. Check your switch documentation to learn more.

You may be wondering to yourself, okay, he's explained what all of this agent stuff is, but what does it give me that I didn't already have? I'm not a cable modem provider, what's in it for me? Excellent question. Let's relate it back to my use case.

Option 82 allows me to say the request that arrived tagged with switch name rack201-sw on port 32 will be given IP address 10.10.1.9. Or, the device that requested an IP and bootfile on port 16 of mgmtswitch1 is going to be rack216-sw with IP 10.10.1.1 and should bootstrap its configuration and self-configure onto the network with all VLANs, IP addresses, and spanning tree configurations predefined.

What Happens When You Can't Get There From Here?

Finally, we get to the part where isc-dhcpd falls short, after a very protracted back story. ISC is tenaciously tied to the notion that you are going to assign an IP address based upon MAC address or assign a dynamic address. It has support for Option 82, but only in a hobbled fashion.

Here are a few interesting implementation details about the isc-dhcpd configuration and its limitations:

- Classes are a way to group attributes:
  - Classes are implemented as a single linked list data structure.
  - Classes can be used in the host block to decide whether to assign an address.
  - Classes cannot be nested.
- Sub-classes present another way to group attributes:
  - Sub-classes are stored efficiently in a hash.
  - Sub-classes can be instantiated dynamically at runtime as an arbitrary group of attributes pulled from the DHCP request (class followed by sub-class).
  - Sub-classes cannot be used in a host attribute to assign an address.
  - Some people have devised source code patches to make sub-classes useful in the host stanza, but they are against old source code and not integrated into the core.
- The host-identifier option can be used to specify "use this other thing as a MAC equivalent." This would appear to be a way to get there, however:
  - You can concatenate various objects dynamically in an isc-dhcpd stanza, like remote ID and circuit ID, but objects created this way cannot be used as host identifiers. They are useful for logging, but not allocation.
  - You can specify that the host-identifier is either a remote device ID or a remote circuit ID, but not both nor in combination.
    - Remote ID is not useful by itself because it's just the relay agent device forwarding the request. You can't make any useful determination based upon that, unless it has only one device plugged into it (you can see why this could be okay for DSL and cable modems).
    - Circuit ID is useful as long as you only have one switch in your entire network acting as a relay agent.
    - To be effective, the IP must be a combination of the two.
  - You can use an isc-dhcpd hack that says "give this requesting device a dynamic IP address in this particular range," where the range is something like 192.168.1.2–192.168.1.2. However, without a way to combine this with circuit ID and remote ID, it has limited use.

Here is my summary about why I find this so disappointing:

We could possibly use classes, if we could concatenate the Remote ID and Circuit ID together to make a unique client designator, but classes are inefficient at that scale, and you cannot use concatenated objects in such a way.

We could possibly use the host identifier, if isc-dhcpd let you use concatenated/generated objects.

We could possibly use the dynamic sub-class spawning facility (remote ID = class, remote-circuit = subclass), but you cannot use sub-classes in host stanzas.

Our Solution

Frustrated and stymied, we went on an isc-dhcpd-alternatives discovery trek. We found a basic Ruby DHCP implementation and extended it to fully support the notion that IP addresses were to be given based upon the remote ID and circuit ID presented by the relay agent. Also, because we have all of the information about all of our hosts in a MySQL database, we extended the relevant network table to include remote ID and circuit ID fields. The Ruby daemon contacts the database upon receiving a query for a given remote-circuit combination, and then serves the host an IP address, network accoutrements, and bootfile, if necessary. This allows us to do complete zero-configuration installation of a switch for the impending supercomputer rollout in 2014, about the time this article will be published.

Furthermore, this implementation has led us down the natural path of considering this for racks of servers. The integrator can save many hours of work collecting and collating MAC addresses because we no longer have to care. We can plug in the switch, let it configure itself, then turn on all of the hosts. They will all get a name and address according to their position on the switch, which dictates their position in the rack. The DNS is preconfigured. Just before publication we deployed two racks of 144 machines each using this DHCP server to assign addresses based upon switch and port. It was a happily successful proof-of-concept.

One last feature of the server is that we can log the MAC address heard in the DHCP request into a MySQL table associated with the network entry for the host. This makes it easy to determine if a given host has moved from one switch to another, or where a particular MAC address lives if we happen to lose track of it. It's a built-in history mechanism and will allow us to track a device that may have been returned for repair to the vendor and then put back into service as a different node later.

The DHCP server configuration file is very simple. It consists of things like the database table, host, username, password, what logging level to use, the interface to bind to, and any site- or network-specific overrides for testing. All of the logic about IP addresses, ports, subnet masks, hostnames, etc. lives in the database, where it belongs. There is essentially nothing to distribute for multiple redundant servers.

Summary

We can have redundant MySQL databases and redundant servers without the need to worry about generating or keeping DHCP configuration files up to date. We can leverage our inventory database (along with minor extension) with our DHCP address allocation and also keep track of the motion of assets that have been repurposed or repaired. Although I have not yet published the source code for this server anywhere, I can possibly share the code upon request. Some things in it are still a bit site specific (like the database table schema), and it so far has only been tested against Dell/Force10 and HP Procurve switches as relay agents. Some documentation cleanup and further testing is in progress.

Lastly, DHCP options are quite powerful. You may find an interesting gem in there that you didn't expect if you take a moment to review them. Option 82 is useful whatever DHCP server you use.

Epilogue

Just before sending this article for final publication, ISC released a new alpha version of isc-dhcpd that reportedly allows one to make an Option 82 determination in a class that includes both the remote ID and circuit ID, something like this:

```
class "1-2-3-9" {
        match if option agent.circuit-id = "1.21.1.4/Ethernet9";
}
```

The remote ID and port get separated by a /. Unfortunately, we were not able to test whether this works. If so, this would reduce the major shortcoming of needing something that can actually deal with both remote ID and circuit ID at the same time and would just leave the burden of generating and distributing a very large configuration file with every host defined in it, and the inherent scalability concern of a single large linked list with every possible combination of switch and port.

References

[1] BOOTP: http://www.ietf.org/rfc/rfc951.

[2] DHCP: http://www.ietf.org/rfc/rfc2131 (obsoletes 1541, updated by 3396, 4361, 5494, 6842).

[3] PXE: http://en.wikipedia.org/wiki/Preboot_Execution_Environment.

[4] DHCP with TFTP: http://www.ietf.org/rfc/rfc1350 (obsoletes 783, updated by 1782, 1783, 1784, 1785, 2347, 2348, 2349).

[5] PXELINUX: http://en.wikipedia.org/wiki/SYSLINUX.

[6] http://www.iana.org/assignments/bootp-dhcp-parameters/bootp-dhcp-parameters.xhtml.

[7] ISC DHCP: https://www.isc.org/downloads/dhcp.

COLUMNS

Practical Perl Tools

MongoDB Meet Perl

DAVID N. BLANK-EDELMAN

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

Mongo just pawn in great game of life.

—As spoken by Alex Karras in *Blazing Saddles*

I had such fun writing my previous column on using Redis from Perl that I thought we should invite another *NoSQL* package to come play with Perl this issue. The package I have in mind, just in case the Mel Brooks reference wasn't clear, is MongoDB (www.mongodb.org).

We will use approximately the same example data set we used in the Redis column for this one as well, but I'm going to do everything I can to avoid making comparisons between the two packages. They do have some small overlapping characteristics (e.g., I will talk about key-value pairs here, too), but they really have quite different mental models so I don't want to do either the disservice by comparing them. One thing that will definitely be different from the past column is that I won't be using the included command-line tool for much of the example output. MongoDB has a similar tool, but it is heavily skewed towards JavaScript (not that that's necessarily a bad thing, but it adds another level of detail that will get in the way). Instead, you'll see more Perl code right up front from me that uses the Perl module called "MongoDB." There are other clients (e.g., *Mango* by the author of Mojolicious), but the standard one is a good place to start.

Think Documents

In the intro I tried to distance MongoDB from Redis, but an even more important disconnect that you'll have to make in your brain comes from the last two letters in the name. If you grew up like I did thinking things that ended in DB are relational databases that want you to talk SQL at them, your first challenge with MongoDB will be to leave a whole bunch of preconceptions behind. Truth be told, even now as I use it, I can't tell if I buy their way of doing things, but let me show it to you and see what you think.

MongoDB is a *document-based* database package. Before you flash to a Microsoft Word file in your head, let me show you what they mean by document:

```
{
   "name" : "USENIX",
   "address" : "2560 Ninth Street, Suite 215"
   "state" : "CA",
   "zip" : "94710",
   "board" : [
    "margo",
    "john",
    "carolyn",
    "brian",
    "david",
    "niels",
    "sasha",
    "dan"
   ],
}
```

This JSON-looking thing (it is actually a format called BSON) contains a bunch of key-value pairs where the values can be different things such as strings, arrays, dates, and even sub-structures of key-value pairs (which they call embedded documents). If this reminds you even a little bit of Perl hash data structures, that's a thought you should water until it grows because we are indeed heading in that direction in a moment.

Documents like the one above are stored in MongoDB collections. Collections are stored in namespaces called "databases." I'm sort of loathe to say this because I think it reinforces bad preconceptions, but their beginner docs make these comparisons: in theory, you could compare MongoDB documents to relational database rows, collections to tables that hold those rows, and databases to, well, databases that contain those tables. But don't do that. I'll explain later why this analogy breaks, so forget I said anything.

Let me see if I can help subvert the dominant paradigm for you: MongoDB has no built-in "joins." MongoDB has no built-in transactions. MongoDB does not enforce a schema for what can be stored. Yes, you can fake all of these things in your application, but MongoDB doesn't have them built-in like (pick your favorite relational database) does. MongoDB means to provide a system that is very fast, very flexible, and very easy to use (with a different mindset) for cases where the above things aren't crucial to your needs. Let's take a look at it now.

Basic Stuff

I guess the first question is how do we actually get a document into MongoDB. Here's the start at a simple Perl answer:

```
use MongoDB;

my $client = MongoDB::MongoClient->new;

my $db = $client->get_database('usenix');
my $org = $db->get_collection('organization');
```

To start, we load the MongoDB module and then create a new client connection. By default, ->new creates a connection to the server running on "localhost" on the standard MongoDB port (27017). These defaults work just peachy for the server I happen to be running on my laptop right now. From there, we indicate which database we want to use and then which collection in that database we'll be working with. At this point, we are ready to do our document insert.

Was that a klaxon I heard? Indeed, here is one of the first places where I expect your cognitive dissonance to spike around MongoDB. Are you perhaps thinking, "Hey, you must have left out a step here! You know, the one where you created the collection, or at least the database? Don't you have to do that before you can write something to them?" Nope. Similar to the way variables, data structures, etc. auto-vivify in Perl, all you have to do is reference something in MongoDB that doesn't exist and "poof," it is now available and ready for you. Is this a good thing? I know I'm not so sure.

Okay, so congratulations, we've created a new database and collection and it is time to perform the insert:

```
my $id = $org->insert(
   {   'name'    => 'USENIX',
       'address' => '2560 Ninth Street, Suite 215',
       'state'   => 'CA',
       'zip'     => '94710',
       'board'   => [qw(margo john carolyn brian david niels
sasha dan)],
   }
);
```

Basically, we just feed a Perl hash data structure to insert() and we're done. There's one subtle thing going on here that isn't apparent because I've chosen to use the insert() defaults. Each document in a collection has a unique ID stored in a field named _id. If we don't choose to make up our own _id when inserting a document, MongoDB will automatically use the ObjectID of the document in that field for us. The return value of insert() is actually the _id of the document.

I stripped the _id field in the document example I gave earlier just because it looks kinda icky, and I didn't want to explain it until now. The real document had a field that looked like this:

```
"_id" : ObjectId("52e9c4565bfcc3d832000000"),
```

MongoDB has a bulk_insert command that is more efficient for multiple document inserts. It takes a reference to an array containing a bunch of hashes. Here's an example:

```
use MongoDB;

my $client = MongoDB::MongoClient->new;

my $db = $client->get_database('usenix');
my $lisa = $db->get_collection('lisaconference');

my @ids = $lisa->batch_insert(
   [   { '2014' => 'Seattle' },
       { '2015' => 'D.C.' },
       { '2016' => 'Boston' },
       { '2017' => 'San Francisco' },
   ]
);
```

It returns a list of _ids.

Find Me

We can prove that the _id field gets added by dumping the contents of all of the documents in the second collection we populated above:

```
use MongoDB;
my $client = MongoDB::MongoClient->new;
my $db = $client->get_database('usenix');
my $lisa = $db->get_collection('lisaconference');

my $lisacursor = $lisa->find();
while ( my $year = $lisacursor->next ) {
   print "---\n";
   while ( my ( $key, $value ) = each %{$year} ) {
      print "$key = $value\n";
   }
}
```

Here's the result:

```
---
_id = 52e9c4565bfcc3d832000001
2014 = Seattle
---
2015 = D.C.
_id = 52e9c4565bfcc3d832000002
---
2016 = Boston
_id = 52e9c4565bfcc3d832000003
---
2017 = San Francisco
_id = 52e9c4565bfcc3d832000004
```

The key thing about this code is that it introduces the find() method. NoSQL (and SQL) databases are really cool and all that, but only if you can actually retrieve the data you put in. The find() method is our primary way for doing this. This code demonstrates a few things about find() and how it works:

1. find() without any arguments will find "everything."
2. find() returns a cursor (this term exists in other database contexts). Think of a cursor as a file handle or iterator that you repeatedly ask for the next data result until it runs out (at which point it returns undef).

In the code above, you can see we used the Perl each() function to be able to pull all of the fields found in the document. If it seems weird from a programmer's point of view that I'm not doing a set of specific hash lookups with known field names (i.e., columns), congratulations, you've hit another place your relational database assumptions don't apply to MongoDB.

In a relational database, you know that if a row has a certain column, all of the other rows in the same table have that column too as part of the table definition. Not true here, my friend. Individual MongoDB documents can include or not include any fields they'd like. Two documents in the same collection can contain or omit any field they'd like.

Now, in practice your application will be inserting a known set of fields into each document (and/or you'll know if you consider certain fields to be optional, so not having them in a document won't be a big surprise). To get any reasonable performance, you'll want indices on known shared fields. So, not total anarchy, just, um, more flexibility than you are used to.

The other thing I should also probably cop to is creating a collection with documents that forced the issue by having no field names (besides the default _id) in common. It's not really a design you'd expect to see used in real life. Let's rejigger things and use a better design for that collection:

```
#... same new, get_database(), get_collection()
# we'll learn about this command in a moment
$lisa->drop;

my @ids = $lisa->batch_insert(
    [     { 'year' => '2014', 'location' => 'Seattle' },
          { 'year' => '2015', 'location' => 'D.C.' },
          { 'year' => '2016', 'location' => 'Boston' },
          { 'year' => '2017', 'location' => 'San Francisco' },
    ]
);
```

So now our data set looks like this (minus the _id fields):

```
---
year = 2014
location = Seattle
---
year = 2015
location = D.C.
---
year = 2016
location = Boston
---
year = 2017
location = San Francisco
```

Now back to find(): more specific queries are made by including a filter when calling find(). Like almost everything else, a filter is specified using a hash. So if we wanted to query all of the documents in our collection for the conferences held in Boston, we'd write:

```
my $lisacursor = $lisa->find({'location' => 'Boston'});
while ( my $conf = $lisacursor->next ) {
      print "$conf->{'year'} is in $conf->{'location'}\n"; }
```

and it would say "2016 is in Boston".

This syntax actually means two things in this context. If we are querying a field whose value is a string, it does a string match as expected. If we are querying a field where the value is an array, like the field "board" in our first document example, MongoDB will find the documents where our filter value is one of the array elements (i.e., it tests for membership). For extra spiffiness, it is possible to use regular expressions, like so:

```
my $lisacursor = $lisa->find({'location' => qr/San/});
```

MongoDB's filter syntax can do more than straight matches. If you use special keywords that start with dollar signs (sorry, Perl programmers!), you can do all sorts of comparisons, like:

```
# find all years < 2017
my $lisacursor = $lisa->find({'year' => {'$lt' => 2017}});
```

or

```
# find all years > 2014 and < 2017
my $lisacursor
   = $lisa->find(  { 'year' =>
                   { '$gt' => 2014, '$lt' => 2017 } } );
```

The filter language is pretty rich, so I recommend you check out the docs for further examples. If you are getting the sense that you won't be mourning the loss of SQL in MongoDB but rather will be translating from the SQL you already know to the MongoDB query syntax you don't, I think that's a reasonable assumption.

I want to mention one more thing before we leave our quick skim of what find() can do: as with SQL, it is often more efficient to specify just which fields you want returned vs. asking for the whole document. MongoDB (like other databases) calls this a *projection*. This is indicated by another hash passed as the second argument to the find():

```
my $lisacursor = $lisa->find( {},
{    'location' => 1,
        '_id' => 0 } );
while ( my $conf = $lisacursor->next ) {
   print "$conf->{location}\n"; }
```

Here you can see I've asked for all documents ({}) and specified that I only want the location field (I explicitly have to request that we don't get _id). The code above yields this lovely list:

```
Seattle
D.C.
Boston
San Francisco
```

Change Is Gonna Come

We've seen the insert() operation; let's talk about how we change things. To get the easiest thing out of the way, you can nuke an entire document using remove():

```
# beware! remove with no filter, i.e., ({})
# removes all documents in that collection, so beware
$lisa->remove({'year' => 2014});
```

To get rid of a collection or a database, there is a similar drop() command as I demonstrated above.

Now on to the fun stuff. To change the contents of a document that exists, there is an update() command of this form:

```
$collection->update({filter},{change},{optional parameters})
```

To begin, we specify what to change and then what change we want to make. The way we specify what change we want to make is akin to the filter examples above. We use special keywords that begin with dollar signs to specify the kind of update. For example, to set a field in a document to a specific value:

```
$lisa->update(
   { 'location' => 'Boston' },
   { '$set' => { 'year' => 2018 } },
);
```

If we want to work with array values, we can use keywords like this:

```
$org->update({'name' => 'USENIX'},
                 {'$pop' => {'board' => 1}});
```

I mentioned optional parameters above. There is an "upsert" parameter that we could add to any of these statements that will change a document if one is found or insert a new one if it not (i.e., "update + insert"). A second parameter worth knowing is the "multi" parameter. With "multi" in place, the change will be made to all documents that match the filter. This is analogous to the bulk-replace functionality you are used to using with UPDATE statements in SQL.

What Else?

We're about out of time here, but, golly, there's a whole bunch of other stuff MongoDB can do. It has aggregation commands, both traditional, like grouping (only spiffier because this can be done as a pipeline), and newfangled (like map-reduce). We can create and adjust indices for better performance. There are replication capabilities and fairly complex sharding configurations. I highly recommend you check out the MongoDB documentation site at docs.mongodb.org to get the full scoop.

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

A Pragmatic Guide to Python 3 Adoption

DAVID BEAZLEY

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

Believe it or not, it's been more than five years since Python 3 was unleashed on the world. At the time of release, common consensus among Python core developers was that it would probably take about five years for there to be any significant adoption of Python 3. Now that the time has passed, usage of Python 3 still remains low. Does the continued dominance of Python 2 represent a failure on the part of Python 3? Should porting existing code to Python 3 be a priority for anyone? Does the slow adoption of Python 3 reflect a failure on the part of the Python developers or community? Is it something that you should be worried about?

There are no clear answers to any of these questions other than to say that "it's complicated." To be sure, almost any discussion of Python 3 on the Internet can quickly turn into a fiery debate of finger pointing and whining. Although, to be fair, much of that is coming from library writers who are trying to make their code work on Python 2 and 3 at the same time—a very different problem than that faced by most users. In this article, I'm going to try and steer clear of that and have a pragmatic discussion of how working programmers might approach the whole Python 3 puzzle.

This article is primary for those who use Python to get actual work done. In other words, I'm not talking about library and framework authors—if that applies to you and you're still not supporting Python 3, stop sitting on the sidelines and get on with it already. No, this article is for everyone else who simply uses Python and would like to keep using it after the Python 3 transition.

Python 3 Background

If you haven't been following Python 3 very closely, it helps to review a bit of history. To my best recollection, the idea of "Python 3" originates back to the year 2000, if not earlier. At that time, it was merely known as "Python 3000"—a hypothetical future version of Python (named in honor of Mystery Science Theater 3000) where all of the really hard bugs, design faults, and pie-in-the-sky ideas would be addressed someday. It was a release reserved for language changes that couldn't be made without also breaking the entire universe of existing code. It was a stock answer that Guido van Rossum could give in a conference talk (e.g., "I'll eventually fix that problem in Python 3000").

Work on an actual Python 3000 version didn't really begin until much later—perhaps around 2005. This culminated in the eventual release of Python 3.0 in December 2008. A major aspect of Python 3 is that backward-incompatible changes were made to the core language. By far, the most visible change is the breakage of the lowly print statement, leading first-time Python 3 users to type a session similar to this:

```
 >>> print "hello world"
   File "<stdin>", line 1
         print "hello world"
                           ^
SyntaxError: invalid syntax
 >>>
```

This is easy to fix—simply change the print statement to `print("hello world")`. However, the fact that even the easiest example breaks causes some developers to grumble and come away with a bad first impression. In reality, the internal changes of Python 3 run much deeper than this, but you're not likely to encounter them as immediately as with `print()`. The purpose of this article isn't to dwell on Python 3 features, however—they are widely published [1] and I've written about them before [2].

Some Assumptions

If you're using Python to solve day-to-day problems, I think there are a few underlying assumptions about software development that might apply to your work. First, it's somewhat unlikely that you're concerned about supporting every conceivable Python version. For example, I have Python 2.7 installed on my machine and I use it for a lot of projects. Although I could enter a time machine and install Python 2.3 on my system to see if my code still works with it, I honestly don't care. Seriously, why would I spend my time worrying about something like that? Even at large companies, I find that there is often an "official" Python version that almost everyone is using. It might not always be the latest version, but it's some specific version of the language. People aren't wasting their time fooling around with different interpreter versions.

I think a similar argument can be made about the choice between Python 2 and 3. If you've made a conscious choice to work on a project in Python 3, there is really no good reason to also worry about Python 2. Again, as an application programmer, why would I do that? If Python 3 works, I'm going to stick with it and use it. I've got better things to be doing with my time than trying to wrap my brain around different language versions. (To reiterate, this is not directed at grumpy library writers.)

Related to both of the above points, I also don't think many application programmers want to write code that involves weird hacks and non-idiomatic techniques—specifically, hacks aimed at making code work on two incompatible versions of the Python language. For example, if I'm trying to use Python to solve some pressing problem, I'm mostly just concerned with that problem. I want my code to be nice and readable—like the code you see in books and tutorials. I want to be able to understand my own code when I come back to read it six months later. I don't want to be sitting in a code review trying to explain some elaborate hacky workaround to a theoretical problem involving Python 2/3 compatibility.

Last, but not least, most good programmers are motivated by a certain sense of laziness. That is, if the code is working fine already, there has to be a pretty compelling reason to want to "fix" it. In my experience, porting a code base to a new language version is just not that compelling. It usually involves a lot of grunt work and time—something that is often in short supply. Laziness also has a dark side involving testing. You know how you hacked up that magic Python data processing script on a Friday afternoon three years ago? Did you write any unit tests for it? Probably not. Yes, this can be a problem too.

So, with the understanding that you probably just want to use a single version of Python, you don't want to write a bunch of weird hacks, you may not have tests, and you're already overworked, let's jump further into the Python 3 fray.

Starting a New Project? Try Python 3

If you're starting a brand new project, there is no reason not to try Python 3 at this point. In fact, it doesn't even have to be too significant. For example, if you find yourself needing to write a few one-off scripts, this is a perfect chance to give Python 3 a whirl without worrying if it will work in a more mission critical setting.

Python 3 can be easily installed side-by-side with any existing Python 2 installation, and it's okay for both versions to coexist on your machine. Typically, if you install Python 3 on your system, the `python` command will run Python 2 and the `python3` command will run Python 3. Similarly, if you've installed additional tools such as a package manager (e.g., setuptools, pip, etc.), you'll find that the Python 3 version includes "3" in the name. For example, `pip3`.

If you rely on third-party libraries, you may be pleasantly surprised at what packages currently work with Python 3. Most popular packages now provide some kind of Python 3 support. Although there are still some holdouts, it's worth your time to try the experiment of installing the packages you need to see if they work. From personal experience over the last couple of years, I've encountered very few packages that don't work with Python 3.

Once you've accepted the fact that you're going to use Python 3 for your new code, the only real obstacle to starting is coming to terms with the new `print()` function. Yes, you're going to screw that up a few hundred times because you're used to typing it as a statement out of habit. However, after a day of coding, adding the parentheses will become old hat. Next thing you know, you're a Python 3 programmer.

What To Do with Your Existing Code?

Knowing what to do with existing code in a Python 3 universe is a bit more delicate. For example, is migrating your code something that you should worry about right now? If you don't migrate, will your existing programs be left behind in the dustbin of coding history? If you take the plunge, will all your time be consumed by fixing bugs due to changes in Python 3 semantics? Are the third-party libraries used by your application available in Python 3?

These are all legitimate concerns. Thus, let's explore some concrete steps you can take with the assumption that migrating your code to Python 3 is something you might consider eventually if it's not too painful, maybe.

Do Nothing!

Yes, you heard that right. If your programs currently work with Python 2 and you don't need any of the new functionality that Python 3 provides, there's little harm in doing nothing for now. There's often a lot of pragmatic wisdom in the old adage of "if it ain't broke, don't fix it." In fact, I would go one step further and suggest that you *NOT* try to port existing code to Python 3 unless you've first written a few small programs with Python 3 from scratch.

Currently, Python 2 is considered "end of life" with version 2.7. However, this doesn't mean that 2.7 will be unmaintained or unsupported. It simply means that changes, if any, are reserved for critical bug fixes, security patches, and similar activity. Starting in 2015, changes to Python 2.7 will be reserved to security-only fixes. Beyond that, it is expected that Python 2.7 will enter an extended maintenance mode that might last as long as another decade (yes, until the year 2025). Although it's a little hard to predict anything in technology that remote, it seems safe to say that Python 2.7 isn't going away anytime soon. Thus, it's perfectly fine to sit back and take it slow for a while.

This long-term maintenance may, in fact, have some upsides. For one, Python 2.7 is a very capable release with a wide variety of useful features and library support. Over time, it seems clear that Python 2.7 will simply become the de facto version of Python 2 found on most machines and distributions. Thus, if you need to worry about deploying and maintaining your current code base, you'll most likely converge upon only one Python version that you need to worry about. It's not unlike the fact that real programmers are still coding in Fortran 77. It will all be fine.

Start Writing Code in a Modern Style

Even if you're still using Python 2, there are certain small steps you can take to start modernizing your code now. For example, make sure you're always using new-style classes by inheriting from object:

```
class Point(object):
    def __init__(self, x, y):
            self.x = x
            self.y = y
```

Similarly, make sure you use the modern style of exception handling with the "as" keyword:

```
try:
   x = int(val)
except ValueError as exc: # Not: except ValueError, exc:
...
```

Make sure you use the more modern approaches to certain built-in operations. For example, sorting data using key functions instead of the older compare functions:

```
names = ['paula', 'Dave', 'Thomas', 'lewis']
names.sort(lambda n1, n2: cmp(n1.upper(), n2.upper()))    # OLD
names.sort(key=lambda n: n.upper())                       # NEW
```

Make sure you're using proper file modes when performing I/O. For example, using mode 't' for text and mode 'b' for binary:

```
f = open('sometext.txt', 'rt')
g = open('somebin.bin', 'rb')
```

These aren't major changes, but a lot of little details like this come into play if you're ever going to make the jump to Python 3 later on. Plus, they are things that you can do now without breaking your existing code on Python 2.

Embrace the New Printing

As noted earlier, in Python 3, the print statement turns into a function:

```
 >>> print('hello', 'world')
hello world
>>>
```

You can turn this feature on in Python 2 by including the following statement at the top of each file that uses `print()` as a function:

```
 from __future__ import print_function
```

Although it's not much of a change, mistakes with `print` will almost certainly be the most annoying thing encountered if you switch Python versions. It's not that the new print function is any harder to type or work with—it's just that you're not used to typing it. As such, you'll repeatedly make mistakes with it for some time. In my case, I even found myself repeatedly typing `printf()` in my programs as some kind of muscle-memory holdover from C programming.

Run Code with the -3 Option

Python 2.7 has a command line switch -3 that can warn you about more serious and subtle matters of Python 3 compatibility. If you enable it, you'll get warning messages about your usage of incompatible features. For example:

```
 bash % python2.7 -3
>>> names = ['Paula', 'Dave', 'Thomas', 'lewis']
>>> names.sort(lambda n1, n2: cmp(n1.upper(), n2.upper()))
__main__:1: DeprecationWarning: the cmp argument is not
supported in 3.x
>>>
```

With this option, you can take steps to find an alternative implementation that eliminates the warning. Chances are, it will improve the quality of your Python 2 code, so there are really no downsides.

Future Built-ins

A number of built-in functions change their behavior in Python 3. For example, zip() returns an iterator instead of a list. You can include the following statement in your program to turn on some of these features:

```
from future_builtins import *
```

If your program still works afterwards, there's a pretty good chance it will continue to work in Python 3. So it's usually a useful idea to try this experiment and see if anything breaks.

The Unicode Apocalypse

By far, the hardest problem in modernizing code for Python 3 concerns Unicode [3]. In Python 3, all strings are Unicode. Moreover, automatic conversions between Unicode and byte strings are strictly forbidden. For example:

```
>>> # Python 2 (works)
>>> 'Hello' + u'World'
u'HelloWorld'
>>>

>>> # Python 3 (fails)
>>> b'Hello' + u'World'
Traceback (most recent call last):
   File "<stdin>", line 1, in
TypeError: can't concat bytes to str
>>>
```

Python 2 programs are often extremely sloppy in their treatment of Unicode and bytes, interchanging them freely. Even if you don't think that you're using Unicode, it still might show up in your program. For example, if you're working with databases, JSON, XML, or anything else that's similar, Unicode almost always creeps into your program.

To be completely correct about treatment of Unicode, you need to make strict use of the encode() and decode() methods in any conversions between bytes and Unicode. For example:

```
>>> 'Hello'.decode('utf-8') + u'World'    # Result is Unicode
u'HelloWorld'
>>> 'Hello' + u'World'.encode('utf-8')    # Result is bytes
'HelloWorld'
>>>
```

However, it's really a bit more nuanced than this. If you know that you're working with proper text, you can probably ignore all of these explicit conversions and just let Python 2 implicitly convert as it does now—your code will work fine when ported to Python 3. It's the case in which you know that you're working with byte-oriented non-text data that things get tricky (e.g., images, videos, network protocols, and so forth).

In particular, you need to be wary of any "text" operation being applied to byte data. For example, suppose you had some code like this:

```
f = open('data.bin', 'rb')     # File in binary mode
data = f.read(32)              # Read some data
parts = data.split(',')        # Split into parts
```

Here, the problem concerns the split() operation. Is it splitting on a text string or is it splitting on a byte string? If you try the above example in Python 2 it works, but if you try it in Python 3 it crashes. The reason it crashes is that the data.split(',') operation is mixing bytes and Unicode together. You would either need to change it to bytes:

```
parts = data.split(b',')
```

or you would need to decode the data into text:

```
parts = data.decode('utf-8').split(',')
```

Either way, it requires careful attention on your part. In addition to core operations, you also must focus your attention on the edges of your program and, in particular, on its use of I/O. If you are performing any kind of operation on files or the network, you need to pay careful attention to the distinction between bytes and Unicode. For example, if you're reading from a network socket, that data is always going to arrive as uninterpreted bytes. To convert it to text, you need to explicitly decode it according to a known encoding. For example:

```
data = sock.recv(8192)
text = data.decode('ascii')

import urllib
u = urllib.urlopen('http://www.python.org')
text = u.read().decode('utf-8')
```

Likewise, if you're writing text out to the network, you need to encode it:

```
text = 'Hello World'
sock.send(text.encode('ascii'))
```

Again, Python 2 is very sloppy in its treatment of bytes—you can write a lot of code that never performs these steps. However, if you move that code to Python 3, you'll find that it breaks.

Even if you don't port, resolving potential problems with Unicode is often beneficial even in a Python 2 codebase. At the very least, you'll find yourself resolving a lot of mysterious `UnicodeError` exceptions. Your code will probably be a bit more reliable. So it's a good idea.

Taking the Plunge

Assuming that you've taken all of these steps of modernizing code, paying careful attention to Unicode and I/O, adopting the `print()` function, and so forth, you might actually be ready to attempt a Python 3 port, maybe.

Keep in mind that there are still minor things that you might need to fix. For example, certain library modules get renamed and the behavior of certain built-in operations may vary slightly. However, you can try running your program through the 2to3 tool and see what happens. If you haven't used 2to3, it simply identifies the parts of your code that will have to be modified to work on Python 3. You can either use its output as a guide for making the changes yourself, or you can instruct it to automatically rewrite your code for you. If you're lucky, adapting your code to Python 3 may be much less work than you thought.

What About Compatibility Libraries?

If you do a bit a research, you might come across some compatibility libraries that aim to make code compatible with both Python 2 and 3 (e.g., "six," "python-modernize," etc.). As an application programmer, I'm somewhat reluctant to recommend the use of such libraries. In part, this is because they sometimes translate code into a form that is not at all idiomatic or easy to understand. They also might introduce new library dependencies. For library writers who are trying to support a wide range of Python versions, such tools can be helpful. However, if you're just trying to use Python as a normal programmer, it's often best to just keep your code simple. It's okay to write code that only works with one Python version.

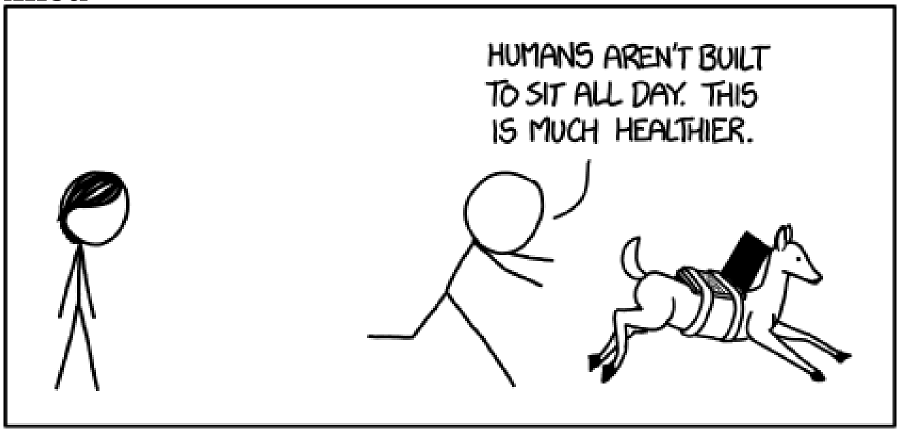
References

[1] Nick Coghlan's "Python 3 Q&A" (http://ncoghlan-devs-python-notes.readthedocs.org/en/latest/python3/questions_and_answers.html) is a great read concerning the status of Python 3 along with its goals.

[2] David Beazley, "Three Years of Python 3," *;login:*, vol. 37, no. 1, February 2012: beazley12-02_0.pdf.

[3] For the purposes of modernizing code, I recommend Ned Batchelder's "Pragmatic Unicode" presentation (http://nedbatchelder.com/blog/201203/pragmatic_unicode.html) for details on sorting out Unicode issues in Python 2 and preparing your mind for work in Python 3.

xkcd xkcd.com



MY HOBBY: ONE-UPPING THE STANDING DESK PEOPLE

iVoyeur
ChatOps

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

Once upon a time, not too long ago, I was locked in a small room with a horde of telephone salesmen. They were quite enamored of VOIP, and SIP phones, and MPLS, and together with one of the executives, they were convinced that telephones (yes, telephones) were the final and ultimate solution to every productivity-related problem in the company.

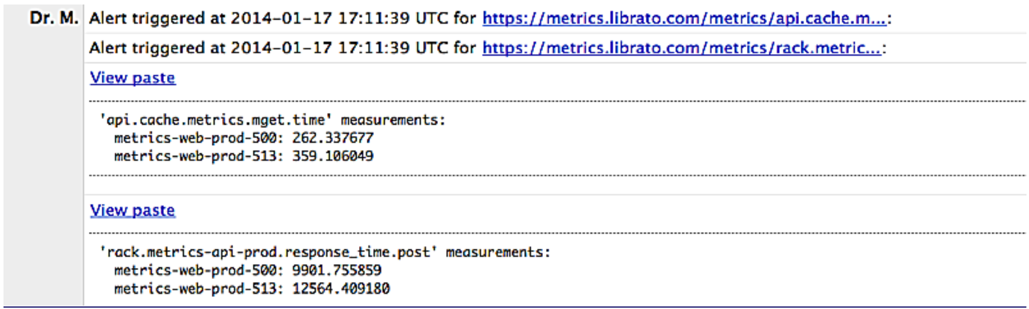
They had a grand vision to unite all workers of the company by attaching them (nearly symbiotically) to a device called a "SIP phone." The SIP phone would sit on everyone's desk and inform everyone of everyone else's status via the blue-hued LED magic of something called a "presence protocol."

Everyone, I was told, would log into their phones first thing every morning, and they would message each other via phone protocols, as well as transfer files to each other this way. Everyone would always know the moment everyone else logged in to work, and because everyone would do everything through their phone, pie charts (yes, pie charts) could be produced that detailed the work habits of the whole company (except of course the execs). The executives really liked pie charts, so they had locked the Ops guys (us) in a small room with the telephone salesmen for six hours (yes, six) to teach us about presence protocols and explain things like how the telephones would replace email and monitoring systems, and how the new system would pay for itself in months once the people were all wired up to the phones. We were evidently expected to do the wiring; honestly, I wasn't looking forward to it.

Meanwhile, on the West Coast, so many companies were in want of smart people that they had snatched up every smart person out there and were actively sending out spies to capture and import more. Some of them had even resorted to stealing smart people from each other, and paying college kids to drop out of school. Other, more respectable, West Coast companies realized that they might be able to use smart people from other parts of the world without shanghaiing them, if they could just figure out how to handle remote workers, so they started exploring the concept of "distributed teams" [1]. Oddly, they collectively came to a vastly different conclusion about the best way to manage their employees—one that did not involve anything like plugging everyone into a phone.

The emergent "distributed teams" phenomenon is based on asynchronous communication, like persistent chat systems, and flextime to maximize individual productivity. Persistent chat is a lot like Web-based IRC, except you get scroll-back of all the conversations that have been going on even while you were disconnected (hence, persistent). These tools integrate easily with other tools and services, especially operations-focused tools, and so they've become a popular solution to centralize operations undertakings, like troubleshooting, and software deployments via chatbot.

Collectively referred to as ChatOps [2], this loosely collaborative alternative to approaches like the phones described above is often referred to in a theoretical way on this blog or that [3, 4] these days, but how it works in practice is not well documented and is something you really



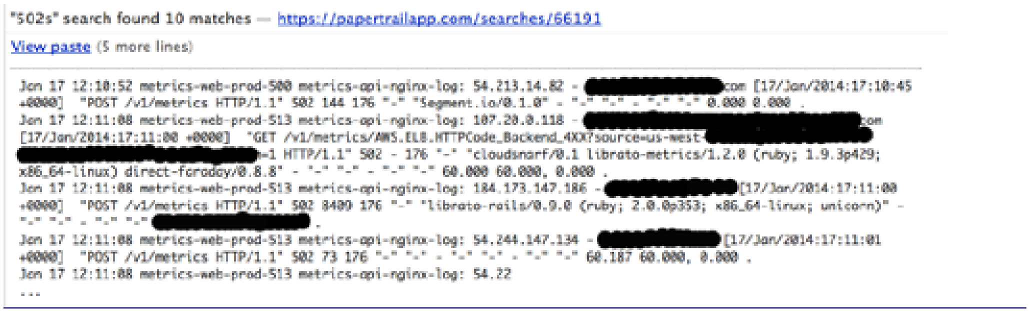
```
Dr. M.  Alert triggered at 2014-01-17 17:11:39 UTC for https://metrics.librato.com/metrics/api.cache.m...:
        Alert triggered at 2014-01-17 17:11:39 UTC for https://metrics.librato.com/metrics/rack.metric...:
        View paste

         'api.cache.metrics.mget.time' measurements:
           metrics-web-prod-500: 262.337677
           metrics-web-prod-513: 359.106049

        View paste

         'rack.metrics-api-prod.response_time.post' measurements:
           metrics-web-prod-500: 9901.755859
           metrics-web-prod-513: 12564.409180
```

Figure 1: Initial notifications of API problems appear in our Ops channel.



```
"502s" search found 10 matches — https://papertrailapp.com/searches/66191
View paste (5 more lines)

Jan 17 12:10:52 metrics-web-prod-500 metrics-api-nginx-log: 54.213.14.82 - [redacted]com [17/Jan/2014:17:10:45
+0000]  "POST /v1/metrics HTTP/1.1" 502 144 176 "-" "Segment.io/0.1.0" - "-" "-" - "-" "-" 0.000 0.000 .
Jan 17 12:11:08 metrics-web-prod-513 metrics-api-nginx-log: 107.20.0.118 - [redacted]com
[17/Jan/2014:17:11:00 +0000]  "GET /v1/metrics/AWS.ELB.HTTPCode_Backend_4XX?source=us-west-[redacted]
[redacted]=1 HTTP/1.1" 502 - 176 "-" "cloudsnarf/0.1 librato-metrics/1.2.0 (ruby; 1.9.3p429;
x86_64-linux) direct-faraday/0.8.8" - "-" "-" - "-" "-" 60.000 60.000, 0.000 .
Jan 17 12:11:08 metrics-web-prod-513 metrics-api-nginx-log: 184.173.147.186 - [redacted][17/Jan/2014:17:11:00
+0000]  "POST /v1/metrics HTTP/1.1" 502 8409 176 "-" "librato-rails/0.9.0 (ruby; 2.0.0p353; x86_64-linux; unicorn)" -
"-" "-" - "-" "-" [redacted] .
Jan 17 12:11:08 metrics-web-prod-513 metrics-api-nginx-log: 54.244.147.134 - [redacted][17/Jan/2014:17:11:01
+0000]  "POST /v1/metrics HTTP/1.1" 502 73 176 "-" "-" - "-" "-" - "-" "-" 60.187 60.000, 0.000 .
Jan 17 12:11:08 metrics-web-prod-513 metrics-api-nginx-log: 54.22
...
```

Figure 2: Follow-up alerts from our log processing system appear in the Ops channel.

need to see in the wild before you comprehend just how nice it is. So, having just joined the distributed workforce, I thought it might be interesting to share a peek inside one such company—how we use ChatOps to communicate, troubleshoot, and monitor our own infrastructure.

My story begins a few days ago when our production API experienced a small glitch. Not the kind of thing that provokes a thorough postmortem, but just a momentary network issue of the sort that briefly degrades service.

Glitches like this are to be expected, but when you're running distributed applications, and especially multi-tenant SaaS, these little hiccups are sometimes the harbingers of disaster—they cannot be allowed to persist and should be detected and investigated as quickly as possible. Our glitch on this particular morning didn't grow into anything disastrous, but I thought it might be nice to share it with you, to give you a peek at what problem detection and diagnosis looks like at Librato.

A Culture of ChatOps

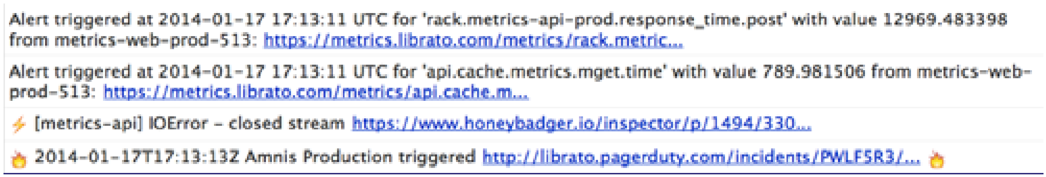
About half of our engineers are remote, so unsurprisingly we rely heavily on ChatOps for everything from diabolical plotting to sportsball banter. Because it's already where we tend to "be" as a company, we've put some work into integrating our persistent chat tool with many of the other tools we use. It should be no surprise, then, that our first hint something was wrong came by way of chat (Figure 1).

Dr. Manhattan (Dr. M), a special-purpose account used for tools integration, is the means by which our various third-party service providers feed us notifications and—like his namesake—can talk to all of our service providers at the same time. In this paste, he's letting us know that he's gotten two notifications from our own alerting feature. The first alert means that our API is taking longer than normal to look up metric names stored in memcached, while the second indicates that our metrics API is responding slowly in general to HTTP POSTs.

Thanks to our campfire [5] integration, Dr. M. is also able to tell us the names of the hosts that are breaking the threshold and their current values. This is alarming enough, but these alerts are quickly followed by more. First comes a notification from our log processing system (Figure 2).

We've configured rsyslog on our AWS hosts to send a subset of our ngnix logs to a third-party alert processing service (Papertrail). About the same time those metrics crossed threshold, Papertrail noticed some HTTP 502 errors in our logs and is sending them to Dr. M. who is listing them in channel. Some of these lines indicate that a small number of requests are failing to post. Not good.

More trouble follows, including several more alerts (Figure 3) relating to our API response time, as well as notifications from our alert escalation service [6], and our exception reporting service [7], the latter of which indicates that some of our users' sessions might be failing out with I/O errors.



Alert triggered at 2014-01-17 17:13:11 UTC for 'rack.metrics-api-prod.response_time.post' with value 12969.483398 from metrics-web-prod-513: https://metrics.librato.com/metrics/rack.metric...

Alert triggered at 2014-01-17 17:13:11 UTC for 'api.cache.metrics.mget.time' with value 789.981506 from metrics-web-prod-513: https://metrics.librato.com/metrics/api.cache.m...

⚡ [metrics-api] IOError – closed stream https://www.honeybadger.io/inspector/p/1494/330...

2014-01-17T17:13:13Z Amnis Production triggered http://librato.pagerduty.com/incidents/PWLF5R3/...

Figure 3: A third set of alerts from various service providers appear in channel.

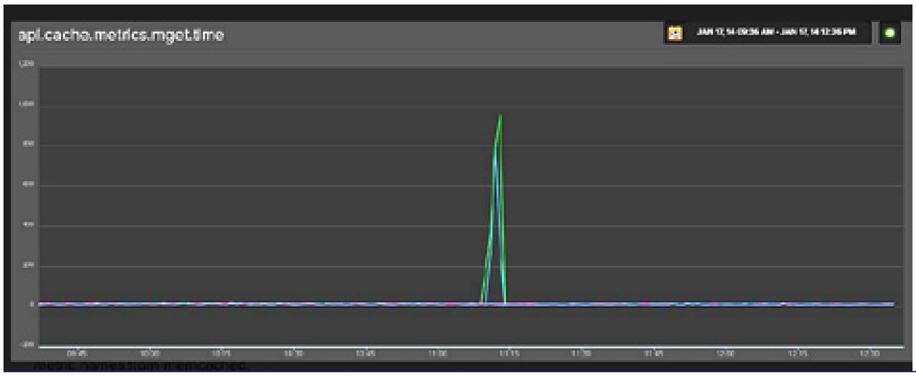
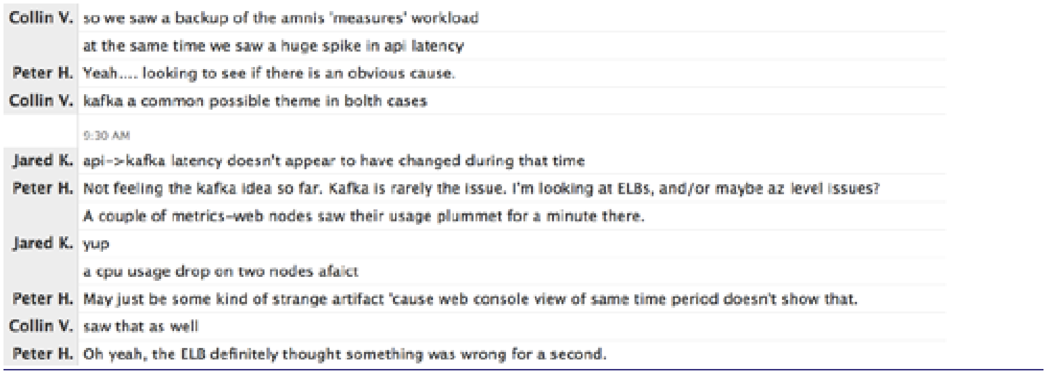


Figure 4: Our API latency is visualized from links in the chat-alerts.

Jared K.	wow

Figure 5: Our engineers react to the influx of bad news.



Collin V.	so we saw a backup of the amnis 'measures' workload
	at the same time we saw a huge spike in api latency
Peter H.	Yeah.... looking to see if there is an obvious cause.
Collin V.	kafka a common possible theme in bolth cases
	9:30 AM
Jared K.	api->kafka latency doesn't appear to have changed during that time
Peter H.	Not feeling the kafka idea so far. Kafka is rarely the issue. I'm looking at ELBs, and/or maybe az level issues?
	A couple of metrics-web nodes saw their usage plummet for a minute there.
Jared K.	yup
	a cpu usage drop on two nodes afaict
Peter H.	May just be some kind of strange artifact 'cause web console view of same time period doesn't show that.
Collin V.	saw that as well
Peter H.	Oh yeah, the ELB definitely thought something was wrong for a second.

Figure 6: Our engineers troubleshoot the latency issue.

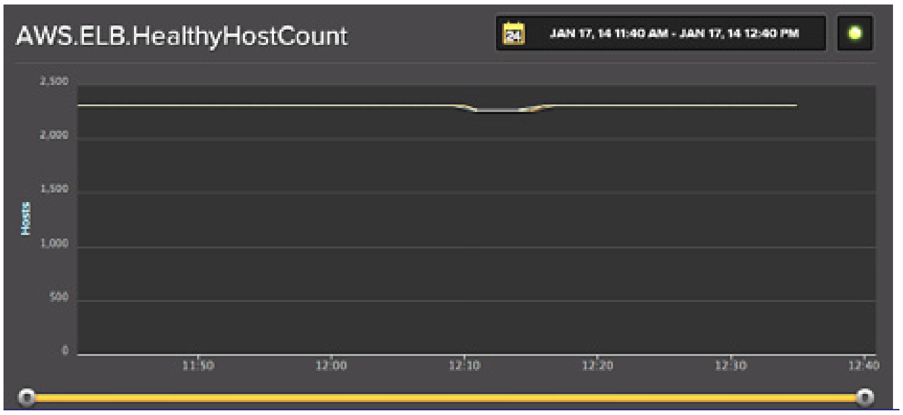
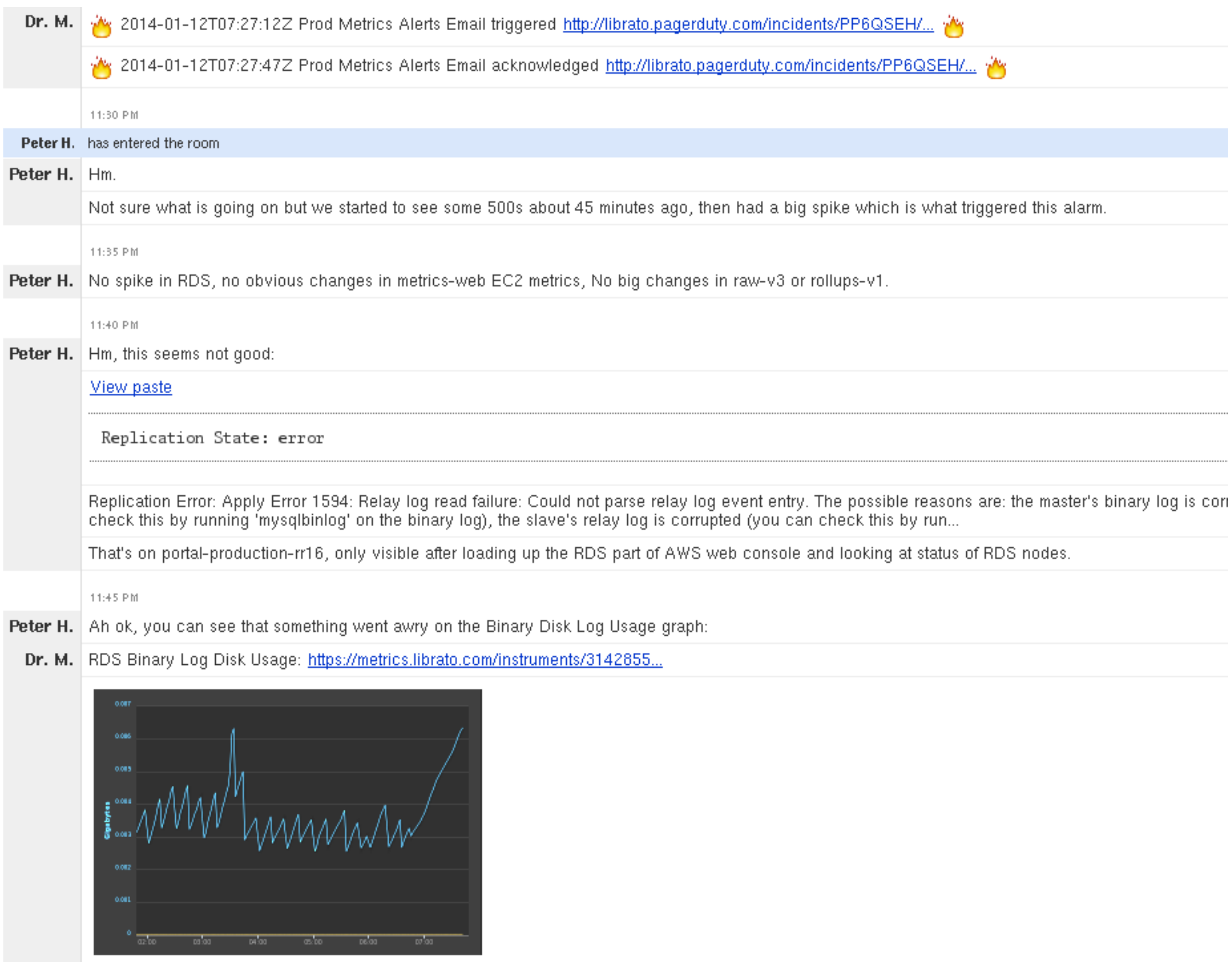


Figure 7: The data confirms a short-lived upstream network partition.

Peter H.

Figure 8: Thumbs up from our operations guys signals the all clear.



Dr. M. 2014-01-12T07:27:12Z Prod Metrics Alerts Email triggered http://librato.pagerduty.com/incidents/PP6QSEH/...

2014-01-12T07:27:47Z Prod Metrics Alerts Email acknowledged http://librato.pagerduty.com/incidents/PP6QSEH/...

11:30 PM

Peter H. has entered the room

Peter H. Hm.

Not sure what is going on but we started to see some 500s about 45 minutes ago, then had a big spike which is what triggered this alarm.

11:35 PM

Peter H. No spike in RDS, no obvious changes in metrics-web EC2 metrics, No big changes in raw-v3 or rollups-v1.

11:40 PM

Peter H. Hm, this seems not good:

View paste

```
Replication State: error
```

Replication Error: Apply Error 1594: Relay log read failure: Could not parse relay log event entry. The possible reasons are: the master's binary log is cor
check this by running 'mysqlbinlog' on the binary log), the slave's relay log is corrupted (you can check this by run...

That's on portal-production-rr16, only visible after loading up the RDS part of AWS web console and looking at status of RDS nodes.

11:45 PM

Peter H. Ah ok, you can see that something went awry on the Binary Disk Log Usage graph:

Dr. M. RDS Binary Log Disk Usage: https://metrics.librato.com/instruments/3142855...

Figure 9: Our operations staff, talking to themselves.

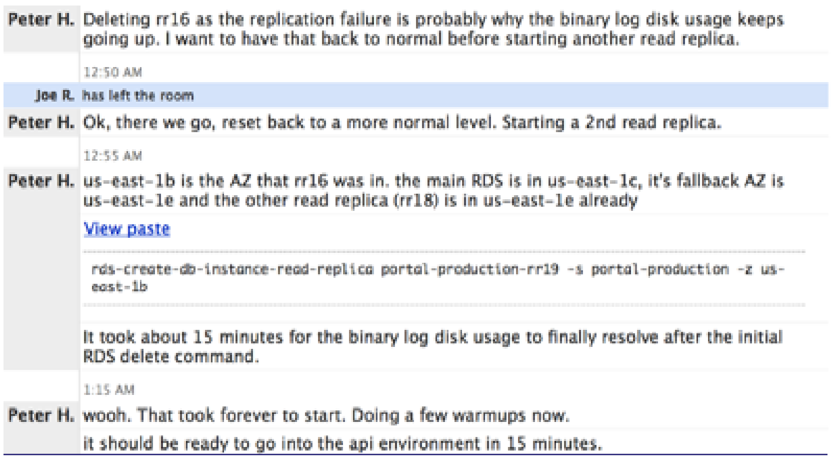
In every case, the notifying entity provides us a link that we can use to get more info. For example, clicking the link in the first notification from our alerting feature yielded the graph (Figure 4) in our metrics UI.

Sure enough, there are two obvious outliers here, indicating that two machines (out of the dozen or so API hosts) were three orders of magnitude slower than their peers returning metric names from memcached.

Our engineers take a moment to assimilate the barrage of bad news that was just dumped into the channel (Figure 5), and then they dig in to figure out just what exactly is going on. Having machine notifications inline with human conversation is a huge win for us. The ability to react directly to what we are seeing in channel without having to context-switch between our phones and workstations makes us a more productive team—everyone is literally on the same page all the time (Figure 6). We win again when the troubleshooting we do as a team is automatically documented, and any newcomers to the channel who join mid-crisis get all the context in the scrollback when they join.

We initially suspected that our message queue might be the culprit, but we were able to quickly check the queue latency graph and eliminate that possibility without wasting time poking at the queue directly. Then we noticed some aberrant system-level stats on the two hosts that broke threshold in the initial alert.

Our metrics tool puts all of our production metrics in one place, so it's trivial to correlate metrics from one end of the stack to the other. Using a combination of application-layer and systems-level graphs, we were able to verify that the problem was in fact



Peter H. Deleting rr16 as the replication failure is probably why the binary log disk usage keeps going up. I want to have that back to normal before starting another read replica.

12:50 AM

Joe R. has left the room

Peter H. Ok, there we go, reset back to a more normal level. Starting a 2nd read replica.

12:55 AM

Peter H. us-east-1b is the AZ that rr16 was in. the main RDS is in us-east-1c, it's fallback AZ is us-east-1e and the other read replica (rr18) is in us-east-1e already

View paste

```
rds-create-db-instance-read-replica portal-production-rr19 -s portal-production -z us-
east-1b
```

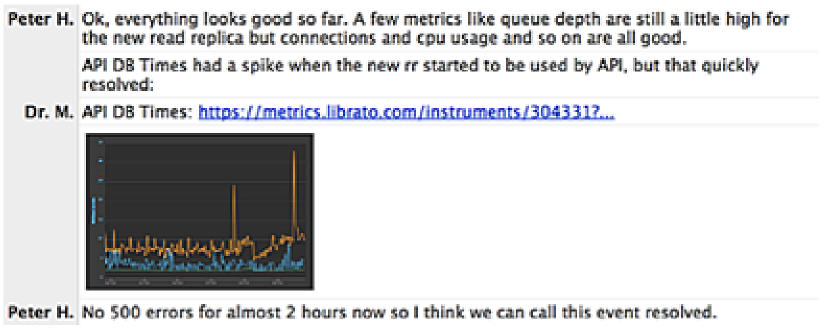
It took about 15 minutes for the binary log disk usage to finally resolve after the initial RDS delete command.

1:15 AM

Peter H. wooh. That took forever to start. Doing a few warmups now.

it should be ready to go into the api environment in 15 minutes.

Figure 10: Our on-call engineer deletes a malfunctioning database replica.



Peter H. Ok, everything looks good so far. A few metrics like queue depth are still a little high for the new read replica but connections and cpu usage and so on are all good.

API DB Times had a spike when the new rr started to be used by API, but that quickly resolved:

Dr. M. API DB Times: https://metrics.librato.com/instruments/304331?...

Peter H. No 500 errors for almost 2 hours now so I think we can call this event resolved.

Figure 11: Our on-call engineer brings up a new replica and monitors the result.

some sort of short-lived network partition that isolated those two particular AWS nodes. The data included the graph in Figure 7, which depicts our total number of healthy AWS nodes (this data sources from AWS CloudWatch [8], using our turnkey AWS integration feature).

The "dip" you see indicates that two of our nodes went dark for a few seconds (too short a length of time for them to have bounced). We noted their names and will keep them under observation for a few days, but at that point all we could do was glare in Amazon's general direction and call "all clear" (Figure 8).

Communicate to Document

We love ChatOps so much that sometimes—late at night, when nobody is around—you can catch our engineers talking to themselves in channel (Figure 9).

This happened not long ago, late at night on a Sunday. Our on-call Ops engineer was alerted by pager via our escalation service about a database problem. You can see both the escalation alert and our engineer's acknowledgment shortly before he joins the channel at the top. As he troubleshoots the issue, he narrates his discoveries and pastes interesting tidbits, including log snippets and graphs of interesting metrics. In this way, he documents the entire incident from detection to resolution for the other engineers who will see it when they join the channel the next morning. Using ChatOps to document incidents has become an invaluable practice for us. Important customer interactions, feature ideas, code deploys, and sales stats are also communicated asynchronously, company-wide, via ChatOps. It is our portal, wiki, and water cooler.

Another practice that we would find it hard to live without is that of sharing graphs back and forth in channel in the way Peter has done above. This is a handy way to both communicate what you're seeing to other engineers and simultaneously document it for later. We'd begun to rely so heavily on copy/pasting graphs to each other that we added a snapshot [9] feature to our metrics tool so our engineers can share the graph they're looking at in our UI directly to a named chatroom in our chat tool without copy/paste.

In this particular incident, Peter tracks the issue down to a read-replica failure on a particular database node, and decides to replace the node with a fresh instance. He first deletes the faulty host and waits for things to normalize (Figure 10), and then he brings up the new node (Figure 11) and monitors dashboards until he's convinced everything is copacetic.

Throughout his monologue, Peter is using the snapshots feature I mentioned earlier to share graphs by clicking on the graph in our UI, and then hitting the snapshot button to send it to Dr. M., who, in turn, pastes it into the channel for everyone to see. Snapshots are generally preferable to copy/pasting because they provide everyone in channel a static PNG of the graph as well as a link back to the live graph in our UI. Those of us who use the Propane [10] client for Campfire even get live graphs [11] in channel instead of boring PNGs.

ChatOps Works

ChatOps delivers on the promise of remote presence in a way the presence protocols never did. It's a natural estuary for stuff that's going on right now; information just can't help but find its way there, and having arrived, it is captured for posterity. ChatOps is asynchronous but timely, brain-dead simple yet infinitely flexible. It automatically documents our internal operations in a way that is transparent and repeatable, and somehow manages to make time and space irrelevant.

Furthermore, because our monitoring is woven into every layer of the stack and is heavily metrics-driven, data is always available to inform our decisions. We spend less time troubleshooting than we would if we chose to rely on more siloed, legacy techniques. Instrumentation helps us to quickly validate or disprove our hunches, focusing our attention always in the direction of root cause.

References

[1] An Ode to Distributed Teams: http://blog.idonethis.com/post/41946033190/an-ode-to-distributed-teams.

[2] ChatOps at GitHub (Rubyufza '13): http://www.youtube.com/watch?v=NST3u-GjjFw.

[3] Distributed teams at buffer: http://joel.is/post/59525266381/the-joys-and-benefits-of-working-as-a-distributed-team.

[4] Distributed teams at Zapier: https://zapier.com/blog/how-manage-remote-team/.

[5] Campfire team collaboration tool: https://campfirenow.com/.

[6] Pagerduty: http://www.pagerduty.com/.

[7] Honeybadger: https://www.honeybadger.io/.

[8] AWS CloudWatch: http://aws.amazon.com/cloudwatch/.

[9] Librato Snapshots: https://metrics.librato.com/product/features.

[10] Propane client for Campfire: http://propaneapp.com/.

[11] Librato-Propane library: https://github.com/librato/librato-propane.

Measure Like You Meant It

DAN GEER AND RICHARD BEJTLICH

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

Richard Bejtlich is chief security strategist at FireEye and a nonresident senior fellow at the Brookings Institution. contact@taosecurity.com

"How did it get so late so soon?"

—Dr. Seuss

Pie charts: managers of security programs create them; their bosses consume them. What slice of corporate computers is patched? What percentage of systems runs antivirus software? What versions of various Microsoft Windows operating systems are present in the environment? Pie charts are the answer to questions like those.

The only real purpose of security metrics is decision support; therefore, we question the utility of pie charts. Statistics on patching, antivirus, and OS distribution are needful, but remember—they are "input metrics." They are a means to an end; they are not the end itself.

Managers devise and operate digital security programs in many forms, sometimes summarizing them in the form of the confidentiality-integrity-availability triad. One goal of a digital security program might well be to protect the organization's data from theft by an intruder. Assuming for the sake of argument that such protection is the organization's primary goal, might it not make sense to measure progress toward that goal?

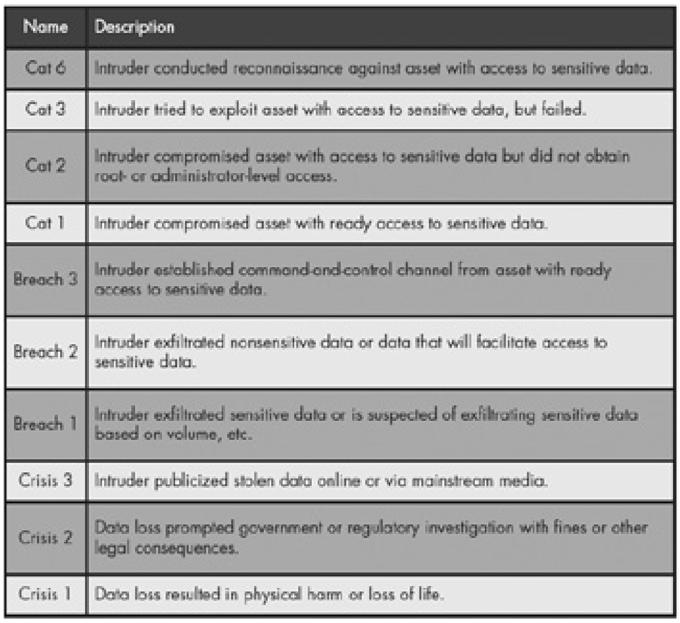
Organizations may well care about mitigating data theft, but seldom do they invest in measuring their progress toward that goal, per se. They spend more time (creating pie charts) on input metrics like patching, antivirus, and OS, but they don't track "output metrics." What they need to ask is, "Are we compromised, and, if so, how bad was the intrusion?" You can spend all the time in the world measuring what goes into the oven, but that won't tell you if you burned the cake—or if a thief stole it.

We argue that the two output metrics for understanding the risk of data theft are (1) counting and classifying digital security incidents, and (2) measuring the time elapsed from the moment of detection to the moment of risk reduction.

1. **Counting and classifying digital security incidents:** An organization should devise a tracking mechanism appropriate for its environment and culture. Security professionals are sure to debate the nature of various intrusions, but don't allow that debate to drag on; you need a taxonomy simple enough to apply and rich enough to usefully describe the incidents likely to be encountered. Figure 1 is a sample set of intrusion categories [1]:

The intrusion "names" given in Figure 1 can easily be replaced with terms tailored to the individual organization.

The focus of this exercise is to count the number of times defensive measures fail, and how badly. A common classification system (emphasis on *common*) is a prerequisite if incident responders are to communicate their true security posture. In Figure 1, they know that if they're facing a Breach 2, they need to implement containment faster than if confronting



Name	Description
Cat 6	Intruder conducted reconnaissance against asset with access to sensitive data.
Cat 3	Intruder tried to exploit asset with access to sensitive data, but failed.
Cat 2	Intruder compromised asset with access to sensitive data but did not obtain root- or administrator-level access.
Cat 1	Intruder compromised asset with ready access to sensitive data.
Breach 3	Intruder established command-and-control channel from asset with ready access to sensitive data.
Breach 2	Intruder exfiltrated nonsensitive data or data that will facilitate access to sensitive data.
Breach 1	Intruder exfiltrated sensitive data or is suspected of exfiltrating sensitive data based on volume, etc.
Crisis 3	Intruder publicized stolen data online or via mainstream media.
Crisis 2	Data loss prompted government or regulatory investigation with fines or other legal consequences.
Crisis 1	Data loss resulted in physical harm or loss of life.

Figure 1: Example of intrusion categories

a Cat 2. Why? Because in the case of a Breach 2, data theft is imminent, whereas a Cat 2 has not yet escalated to the same likelihood of negative consequences. This is what we mean by metrics that deliver decision support.

2. **Measuring the time elapsed from the moment of detection to the moment of risk reduction:** When first reading this output metric, you might ask, "What is risk reduction?" Professional incident responders speak in terms of "containment"—actions taken to remove an intruder's ability to communicate with a compromised resource (i.e., if a compromised computer is "contained," then the intruder can neither steal data from it nor issue remote commands to alter its state). Rogue, independent code that is already on the machine, however, can still take actions whether or not it is cut off from the outside world, such as to scramble or delete data—that risk remains even if the risk of data exfiltration from the compromised system is eliminated when that system can no longer contact its home base (or be contacted by it).

You may ask, "Why measure from the moment of detection to the moment of risk reduction (i.e., containment)? Why not measure time elapsed from the moment of compromise to the moment the resource is returned to a trustworthy state (i.e., recovery)?" Organizations beginning to measure output metrics should start with the more achievable goal of measuring detection-to-containment. On the "left" side of the time horizon, determining when an intrusion first occurred can be a daunting task. In Mandiant's experience, many serious intrusion victims wait months before learning they have been compromised: 243 was the median number of days from compromise to detection, and $^{2}/_{3}$ of the time a third party notified the victim of the intrusion. (Both Verizon's Data Breach Investigations Report and the Index of Cyber Security have found similar numbers.) On the "right" side of the time horizon, moving from containment to recovery is not necessarily the responsibility of the security team; usually the IT team is tasked to rebuild intrusion victims from gold master builds. Measuring that process is outside the security team's purview and can be unnecessarily demoralizing if recovery is a lengthy process.

Or you may ask, "Why does time matter at all? Shouldn't the severity of the intrusion be the most important metric?" Even if it's true that severity is ultimately the most important metric, an incident response team will rarely recognize the severity of an intrusion at the moment of first detection. At best, the team can decide whether the intrusion merits a "fast path" or a "slow path" response process. Using threat intelligence, the most advanced incident response teams identify perpetrators with a history of inflicting the most damage, or having the intention and/or capability of inflicting the most damage. Upon finding these critical threat groups active in the enterprise, the IR team responds using a "fast path," taking actions to quickly implement containment. Other intruders deserve a "slow path." Triage is not just for hospital emergency rooms.

If time is important, what is the reward for being fast? Again, using Mandiant's experience, critical threat groups do not act "at network speed" or "at the speed of light" as is often heard when speaking to government and defense officials. Rather, most data thieves need time to move beyond their initial foothold. They need to find the data they want, figure out a way to remove it, and only then exfiltrate that information. This process may take days or weeks, not nanoseconds. All the while, they must remain unseen by the victim organization and any third parties with extraordinary detection mechanisms. If at any point the defenders detect and interrupt the intruders before they can steal data, the victim organization "wins."

One public example of the role of time comes from the 2012 hack of the State of South Carolina Department of Revenue. As shown in Figure 2, it took the criminal group exactly one month to accomplish its goal [1]. Although the threat actor compromised the agency on August 13, 2012, they did not remove any data until September 13, 2012.

Recalling that $\text{Availability} = \text{MTBF} / \text{MTBF+MTTR}$, where MTBF is "Mean Time Between Failures" and MTTR is "Mean Time To Repair," you can see that 100% availability comes from either making MTBF infinite or making MTTR zero. In the spirit of supporting your decisions, it's clear that there comes a point where further investment in intrusion avoidance is diseconomic. From that point on, if not sooner, your investments should go towards reducing the duration of compromise, hence the very metric we suggest.

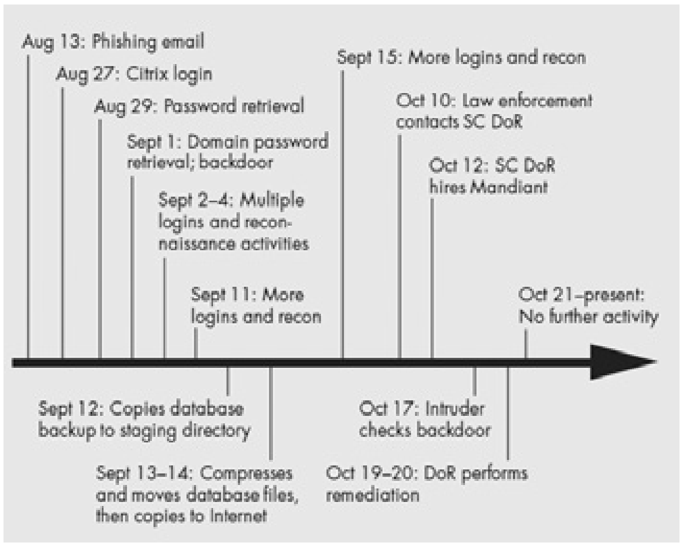

Aug 13: Phishing email
Aug 27: Citrix login
Aug 29: Password retrieval
Sept 1: Domain password retrieval; backdoor
Sept 2–4: Multiple logins and reconnaissance activities
Sept 11: More logins and recon
Sept 15: More logins and recon
Oct 10: Law enforcement contacts SC DoR
Oct 12: SC DoR hires Mandiant
Oct 21–present: No further activity
Sept 12: Copies database backup to staging directory
Sept 13–14: Compresses and moves database files, then copies to Internet
Oct 17: Intruder checks backdoor
Oct 19–20: DoR performs remediation

Figure 2: Timeline of a compromise

While we are at it, if there is no such thing as a "good intrusion" nor is perfect knowledge ever possible, then measuring time is the most concrete way to evaluate incident handling maturity, especially in large organizations suffering many intrusions. Beyond counting/classifying and measuring time as we argue above, we offer one simple indicator that an incident response capability is moving in the right direction: is your organization a member of FIRST? FIRST is the Forum of Incident Response and Security Teams, a global security group founded in 1989 and consisting of nearly 300 members (Figure 3).

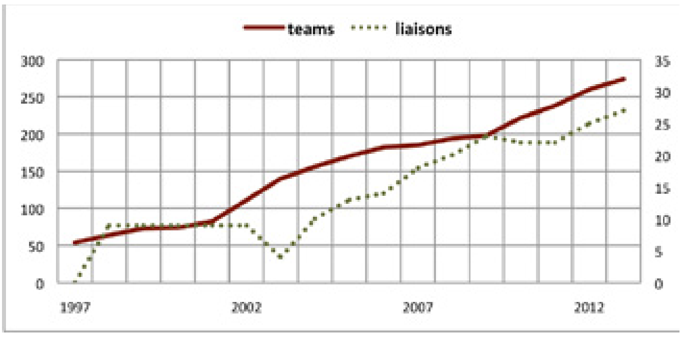


Figure 3: FIRST membership by year

FIRST membership demonstrates a commitment to creating and maintaining a formal incident detection and response program, backed by an audit of a candidate member's capability and recommendation by two current FIRST member teams. By successfully joining FIRST, an organization says, "We are committed to incident response as a core element of a security program." The growth in FIRST membership since 1989 parallels the rise of incident response as a viable security strategy, complementing (and some might say now, partially displacing) incident avoidance.

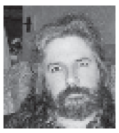
Perhaps we should have admitted it at the outset, but we are pessimists who prefer to think of ourselves as realists. If you are a security program manager, then prepare for when you get hacked next. If you are a statistician, then get the data—you can always throw it away later.

References

[1] Taken from Richard Bejtlich, *The Practice of Network Security Monitoring* (No Starch Press, 2013).

/dev/random

ROBERT G. FERRELL



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

Make no mistake: I miss being a sysadmin. I did not truly realize to what extent that was true until the first job I had where I was a mere user. I couldn't install drivers, write scripts, change system options, put in patches, or much of anything else. To me, a computer represents something you optimize and manage so other people can use it to run programs and print stuff out. In the final analysis, I simply don't *like* being a mere user. It isn't really the sense of power that some sysadmins seem to crave; for me, it's more of an intimacy issue. I've been smitten by computers for all of my adult life and I like to be in touch with the whole enchilada. Being an ordinary user is akin to going to a museum. You can look, but you can't touch. Boring.

Being the system administrator allows you to meld with the computer and feel its innermost heartbeat. You can watch the packets skitter across the network interface, crawl up the stack to the application layer, and then spill out onto the user like so much confetti at a tickertape parade. (Some of you may need to stop and employ an Internet search engine such as Google at this point. I refuse to use "Google" as a verb.)

When you've got root you enter into "with great power comes great responsibility" territory: at once both intoxicating and sobering . . . which I guess would cancel each other out. Another evocative literary sloop dashed against the rocks of semantics. Anyway, sitting at the console with root access is like sitting at the helm of a great starship, the myriad galaxies glittering and twinkling in your forward viewscreen, all awaiting exploration. The process table is your ship's manifest, the processor is the propulsion system, the firewall and IDS are your shields, the router your navigation system, and the weapons bay . . . the weapons bay consists of your brain and your fingers, because if you weren't capable of doing damage that way you probably wouldn't be reading this. I hasten to add that this ship engages in purely defensive actions only. Really.

Your passengers are the users. You must keep them happy by avoiding rough seas (latency), noroviruses (malware), and running out of booze (inadequate quotas). You must also, of course, get them to their destination (process their data successfully) without running aground (crashing the system) or being intercepted and boarded by pirates (see malware, above). If the ship springs a (memory) leak, you must be ready to patch it immediately.

I was going to see how far I could carry the nautical metaphor but I'm starting to feel a little queasy, so it's time to tie off at the nearest dock and disembark. I hope you had a pleasant cruise. Customs should only take three or four hours; don't forget to declare that velvet matador painting and the six liters of tequila and rum you poured into empty shampoo bottles tucked away in your luggage.

I love looking at logs. To me they're like reading the daily news. System logs will show you which processes completed successfully and which took a dive. Security logs reveal who's been rattling your (hopefully locked) gates, casing the joint, or trying to slip malware under the door or through the transom. They can also provide a peek into the secret life of your machine: which applications play well with others and which don't.

Ah, what I wouldn't give for a return to the halcyon days when my mornings consisted of coffee and leisurely log scanning, instead of frantically trying to prioritize the huge stack of overdue tasks by which ones will engender the least abuse from my chain of command when finally completed. Had I known where I would be at this moment 15 years ago, I would have stayed put as a sysadmin. To quote Billy Joel, "if that's movin' up, then I'm movin' out."

Self-pity having had its say, let's get back on task. I must confess to being something of a network voyeur in that I've always been fascinated by using a sniffer to watch packets flitting to and fro. I don't mean reading their payloads so much as just witnessing TCP/IP in action. UDP is less engaging, because there's no flow to it: just the occasional incoming or outgoing datagram, like watching birds patronizing a rather unpopular feeder, or a slow golf game (if that isn't redundant). TCP/IP, on the other hand, is basketball, a continuous stream that experiences peaks and troughs of activity, depending mostly on what's going on at Layer 7.

I've always been the kind who enjoys helping people, so system administration was a natural for me. In contrast to the infamous "BOFH" (I'm using the abbreviation because I'm not sure how the editors would feel about my spelling it out), the powers of the system administrator may also be used for good. Solving problems for users can be very fulfilling. It can also be aggravating, but now that I have spent almost seven years as a user, I understand what makes some users act the way they do.

Speaking of that, sysadmins tend to get the impression that their users are somewhat intellectually challenged. I certainly did. Now that I have quite a bit of experience on the other side, I find myself constantly questioning the cognitive abilities of the IT staff. I suppose that's unavoidable, as neither side has access to the thought processes and priorities of the other. An alternate conclusion one might reach is that I simply think I'm smarter than everyone else, but I'm going to sidestep that one as uncharitable because it's my column and I can do that.

I've heard rumors that the sysadmin is a doomed species whose time on Earth is rapidly dwindling down to a few last chmod spasms. I don't believe them. Taking a human out of the loop may seem to streamline and normalize the administration process, but at least you can reason with a human sysadmin (chocolate and beer, for example, being powerful arguments). When your human-less network server tells you that your quota has been exceeded and your account will be frozen as a result, who you gonna call?

Book Reviews

ELIZABETH ZWICKY, RIK FARROW, AND MARK LAMOURINE

Numbersense: How to Use Big Data to Your Advantage
Kaiser Fung
McGraw Hill, 2013. 218 pages
ISBN 978-0-07-179966-9
Reviewed by Elizabeth Zwicky

This is a fine book about thinking about numbers, only moderately connected to big data. If you really want to know about big data, you'd be better off with one of the recently reviewed data analysis books; only some of the examples here deal directly with big data issues. On the other hand, if you would like a better idea of how data analysts work and how the news is lying to you, without too many actual numbers, this is a nice start. Seriously, a lot of data analysis is more about skepticism than about numerical manipulation more complex than addition and subtraction, and this kind of introduction will move you towards the right thought habits.

Hyperbole and a Half: Unfortunate Situations, Flawed Coping Mechanisms, Mayhem, and Other Things That Happened
Allie Brosh
Simon and Schuster, 2013. 371 pages
ISBN 978-1-4516-6617-5
Reviewed by Elizabeth Zwicky

This is a purely non-technical book, based on a Web comic. It is one of the funniest descriptions of dog ownership ever, and Allie Brosh is one of the few people who can write about depression in ways that are both funny and true. You don't need to take my word for it, you can just go and search for it. In book format you get more text and no video, but the video is not actually required, while the text is excellent.

Although I read the web comic, I had missed some of these, including some laugh out loud stories. Go get a copy so that you can press it on friends you can't email links to, or just so that when your Internet goes out you can while away the time reading and remembering that every so often the Internet brings us marvelous creators, moments of pure hilarity, and utter poignancy. Or, if you like, so that you know that yours is not the stupidest dog ever.

Systems Performance for Enterprise and Cloud
Brendan Gregg
Prentice Hall, 2013;. 735 pages
ISBN 978-0-13-339009-4
Reviewed by Rik Farrow

Although Brendan's book's title refers to performance, the book could just as easily have been called troubleshooting Linux and Solaris systems. And by systems, I do mean everything from caches to distributed services. Brendan writes clearly, leaves nothing out, and is well organized but never boring.

The book begins with four chapters that provide the necessary background for the following eight chapters. Brendan explains, with examples, key concepts, methodology, terminology, kernel internals for the performance analyst, and tools. He uses analogy well: for example, when converting time scales for system latency into human-understandable scales, where if a clock tick is one second, L3 cache access takes 43 seconds, and DRAM access six minutes. The book is full of analogies like this that make the writing easier to comprehend.

The next eight chapters cover particular topic areas, like applications, memory, CPUs, network, and cloud computing, in detail. Brendan designed the book so that it can be used as a reference, and he attempts to future-proof it with the focus on methodology and necessary background. The final chapter contains a lengthy troubleshooting session showing how Brendan uses the methodology that he has described in real life. He ends with a quote from Niels Bohr: "An expert is a person who has made all the mistakes that can be made in a very narrow field." I believe this reflects Brendan's attitude well, in that he speaks from experience and does not talk down to his readers.

There are seven appendices, starting with performance tools for Linux and Solaris and including one of DTrace one-liners. Throughout the book, Brendan shows how tools for both Linux and Solaris-related operating systems are used to display utilization, capacity, saturation, and errors, part of his USE methodology that forms an early step in troubleshooting performance issues.

I found Brendan's writing pleasant to read. The attention to detail is great, and I noticed no typos or mistakes in examples, which was also a real pleasure. I can recommend this book to system designers, system administrators, and programmers because all three groups will benefit by better understanding, and being able to measure, the many subsystems that are important in systems performance. Although this is not a beginner's book, an intermediate to advanced practitioner will get a lot of benefit from reading it, in whole or in part.

RabbitMQ In Action
Alvaro Videla and Jason J. W. Williams
Manning Publications, 2012. 287 pages
ISBN 978-1935182979
Reviewed by Mark Lamourine

RabbitMQ in Action is subtitled "Distributed Messaging for Everyone," and while that's hyperbolic, it's not without a grain of truth. As the timeline in the first chapter notes, for decades, enterprise quality messaging services have been the purview of a handful of commercial providers. In the past 15 years or so, a handful of attempts have been made to bring an open messaging service to distributed applications with mixed results. RabbitMQ and AMQP (Advanced Message Queuing Protocol) show a lot of promise.

I appreciated the context that the authors provide in the introductory chapters. The verbal and graphical timelines for the inception and development of computer messaging are rooted in the financial industry of the early 1980s and carry through an attempt at unification in the JMS standard to the emergence of AMQP from the same world of financial applications in the mid-2000s. The authors also explain how the use of messaging technology improves the robustness and flexibility of distributed applications. This is still the first chapter. The last two pages cover installation and initial configuration of the service.

In the remainder of the book, the authors treat service configuration, clustering, and failover and message store persistence (making sure messages en route aren't lost if the node they're on fails). Several chapters relate to application development, focusing on design patterns and writing for failure. RabbitMQ is written in Erlang but, never fear, all of the code for the examples is in common application languages like Python and PHP. In the final chapters, Videla and Williams touch on managing, monitoring, and securing the RabbitMQ service.

The authors write with a clear flowing style that packs a lot of content into an unimposing book without feeling dense or unnecessarily academic. Videla and Williams have created an accessible introduction to messaging technology in general and to AMQP and RabbitMQ in particular. If you're considering writing a modern distributed application, whether for internal business processes or Web applications, *RabbitMQ in Action* is a good place to start.

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

2014 USENIX Federated Conferences Week

Cloud, Storage, Sysadmin, and More

June 17–20, 2014 Philadelphia, PA **www.usenix.org/fcw14**

USENIX ATC '14: 2014 USENIX Annual Technical Conference

ICAC '14: 11th International Conference on Autonomic Computing

Feedback Computing '14: 9th International Workshop on Feedback Computing

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

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

URES '14: 2014 USENIX Release Engineering Summit NEW!

UCMS '14: 2014 USENIX Configuration Management Summit

WiAC '14: 2014 USENIX Women in Advanced Computing Summit

Training Sessions are back!

Topics include:
- **Autonomic computing**
- **Configuration management**
- **Release engineering**

Registration opens in April. Discounts available!
Register by the Early Bird Deadline, Monday, May 19, 2014, and save.

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

usenix
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION