;login:

OCTOBER 2015 VOL. 40, NO. 5

usenix
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

Operating Systems and Sysadmin

usenix 40TH ANNIVERSARY

usenix
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

UPCOMING EVENTS

LISA15
November 8–13, 2015, Washington, D.C., USA
www.usenix.org/lisa15

Co-located with LISA15:

UCMS '15: 2015 USENIX Container Management Summit
November 9, 2015
www.usenix.org/ucms15

URES '15: 2015 USENIX Release Engineering Summit
November 13, 2015
www.usenix.org/ures15

Enigma
January 25–27, 2016, San Francisco, CA, USA
enigma.usenix.org

FAST '16: 14th USENIX Conference on File and Storage Technologies
February 22–25, 2016, Santa Clara, CA, USA
www.usenix.org/fast16

NSDI '16: 13th USENIX Symposium on Networked Systems Design and Implementation
March 16–18, 2016, Santa Clara, CA, USA
www.usenix.org/nsdi16

Do you know about the USENIX Open Access Policy?

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

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

www.usenix.org/membership

SREcon16
April 7–8, 2015, Santa Clara, CA, USA

USENIX ATC '16: 2016 USENIX Annual Technical Conference
June 22–24, 2016, Denver, CO, USA
Submissions due February 1, 2016
www.usenix.org/atc16

Co-located with USENIX ATC '16:

HotCloud '16: 8th USENIX Workshop on Hot Topics in Cloud Computing
June 20–21, 2016

HotStorage '16: 8th USENIX Workshop on Hot Topics in Storage and File Systems
June 20–21, 2016

SOUPS 2016: Twelfth Symposium on Usable Privacy and Security
June 22–24, 2016
www.usenix.org/soups2016

USENIX Security '16: 25th USENIX Security Symposium
August 10–12, 2016, Austin, TX, USA

Co-located with USENIX Security '16

WOOT '16: 10th USENIX Workshop on Offensive Technologies
August 8–9, 2016

CSET '16: 9th Workshop on Cyber Security Experimentation and Test
August 8, 2016

HotSec '16: 2016 USENIX Summit on Hot Topics in Security
August 9, 2016

OSDI '16: 12th USENIX Symposium on Operating Systems Design and Implementation
November 2–4, 2016, Savannah, GA, USA

LISA16
December 4–9, 2016, Boston, MA, USA

Stay Connected...

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

;login:

OCTOBER 2015 VOL. 40, NO. 5

usenix
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

EDITOR
Rik Farrow
rik@usenix.org

MANAGING EDITOR
Michele Nelson
michele@usenix.org

COPY EDITORS
Steve Gilmartin
Amber Ankerholz

PRODUCTION
Arnold Gatilao
Jasmine Murcia

TYPESETTER
Star Type
startype@comcast.net

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

www.usenix.org

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

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

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

EDITORIAL

Musings

RIK FARROW

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

During the HotCloud '15 workshop, I was invited to join a discussion group. We were supposed to decide which was better, VMs or containers. An hour later, we really hadn't answered the question posed to us, but we did have some answers.

Virtual machine technology has been around since IBM developed VMs as a method for sharing mainframes. That might sound funny, but mainframes in businesses were used to run batch jobs or long-running transaction processing applications, like managing accounts for a bank. Sharing the computer, even if that computer wasn't always busy, was a side issue.

With VMs, customers could run other applications when demand by the main application was low. You could even run IBM's version of UNIX, AIX, and later, Linux, providing the illusion of a time-sharing system that we are most familiar with.

In the early noughts, VM technology really took off. Xen and VMware became popular ways of sharing underused systems. And like the original IBM VMs, you could run applications requiring different operating systems all on the same server.

Containers

Container technology was taking off at about the same time, and the biggest users of containers were companies with clusters all running the same OS. For those uses, running one operating system inside of another, however stripped down, was a waste of processing power. Also, why run an operating system per VM when you could just have a single operating system supporting all of your containers?

For many years, VMs were the prominent technology, with containers being used at Google or hosting companies. And there are both advantages and disadvantages to using containers. While containers were great at improving efficiency and making management easier because there was just one set of system software to manage, containers were not as good as VMs for security. That extra level of separation, eventually supported by special CPU instructions, really did make the combination of a hypervisor and VMs more secure than a system running containers using a single Linux kernel.

And those were, roughly, the results of our discussion group: that VMs were best for running legacy applications and for security, while containers were a packaging framework that is more efficient and easier to manage than VMs. But we did discuss one other technology, one not included in our original remit: unikernels.

The Middle Path

We had two people from Cambridge in our group, and they suggested that we should also consider unikernels, like MirageOS. So let's talk about unikernels.

Where VMs are entire operating systems that happen to be running applications, and containers are namespaces [1] used to isolate just one application, unikernels are more like applications that run directly on top of a hypervisor [2]. Unikernels can be even more efficient than containers because instead of sharing an operating system, like containers, unikernels don't have an operating system. Unikernels by design run a single-threaded application and rely on the hypervisor for access to hardware resources.

You might just be thinking that being constrained to a single thread can be a serious issue, and you'd be right—for some applications. But for many others, a single, stripped down to bare essentials, dedicated thread is just right. Unikernels jettison almost all of the support found in traditional operating systems in exchange for single-minded efficiency.

The unikernel focus on doing one thing has security benefits as well. While VMs and containers include a whole array of applications, such as shells, administrative commands, and compilers, unikernels have nothing except the application and the support library needed by that application for communication with the hypervisor. Unikernels are a manifestation of least privilege and minimal configuration hard to achieve with VMs or containers.

And it turns out that because the security model of containers is weaker than that of VMs or unikernels, most people who use containers run containers belonging to the same security domain within a VM. I was surprised to learn this because it means that most of the gain in performance over VMs gets tossed for stronger security. That containers are still used at all speaks to how much easier it is to manage applications within containers as compared to entire VMs. Someone who works for a company that runs giant clusters mentioned that they even run VMs from within containers, meaning that they start with a VM that runs containers that run VMs. Sounds silly, but the point is that containers are easier to manage than VMs, and that is actually very important to people who run huge clusters.

You might be wondering why we don't see unikernels everywhere, and you are right to wonder. Unikernels appear to be the best choice when it comes to efficiency and security for many applications. But there are some things that the unikernel people aren't going to tell you.

MirageOS, with its Cambridge and Xen connections, is the best known of unikernels today, but there are others: LING, based on Erlang, and HaLVM, based on Haskell, to name two. MirageOS uses OCaml, a functional programming language. Erlang and Haskell are also functional programming languages. Functional programming languages have real advantages when it comes to security, although OCaml does not require the programmer to write purely functional code. Learning how to write in Haskell, for example, requires serious effort on the part of the programmer: you need to think differently, more like a mathematician, to become a useful functional programmer.

The requirement of needing to be a programmer, familiar with functional languages, is currently a huge impediment to the success of unikernels. Unlike VMs, which provide an environment that appears identical to the one that most people normally work with, and with containers, which focus on packaging, working with a unikernel today means using an application written for a particular unikernel technology. You can certainly do that, but you best be a programmer who can adopt the application of your choice to run in that environment.

Perhaps the easiest unikernel technology to use are rump kernels based on NetBSD, as the environment is POSIX and the language commonly used is C. Antti Kantee, one of the primary creators of rump kernels, has written an article in this issue arguing for the use of unikernels. One of his many points is that much of what operating systems provide us with is support needed by time-sharing systems. Time-sharing was a method designed for sharing mainframes among multiple users; today, most servers run applications that provide services, and their users are other applications, not people. Times have changed, but operating systems have remained the same.

Well, I am exaggerating. Operating systems haven't remained quite the same. They have grown. Enormously. For example, Linux has grown from 123 system calls [3] in version 1 to nearly 400 system calls for the 3.2 kernel. Microsoft Windows Server 2012 has 1144 system calls [4]. Operating systems have become incredibly complex.

While researching how to run legacy code securely within Web browsers, Douceur et al. [5] discovered that they could run some desktop applications with minimal modifications while using just a handful of system calls. Unikernels move us closer to a similarly minimal environment.

The Lineup

We start out this issue with an opinion piece by Antti Kantee. While Kantee certainly has his own axe to grind, he also makes some very good points while being amusing at the same time.

Next we have an article about Grappa (no, not the liquor), a distributed shared memory framework developed by a group at the University of Washington, Nelson et al. The Grappa framework creates an abstraction of a single memory space for programmers seeking to develop software that works like Hadoop, Spark, or GraphLab. Their system also hides the latency of remote memory accesses by taking advantage of the parallelism inherent in processing big data.

We next take a look at a different issue, also caused by non-uniform memory access. Lepers et al. studied how the core interconnects work in server-class AMD processors, and discovered that the bandwidth between cores in AMD chips varies tremendously. They developed and tested software that can determine the best placement for multithreaded applications, and migrate threads to cores with more bandwidth between them.

Both of these articles are based on papers presented at USENIX ATC '15. The next article was related to a FAST '15 paper and presents a novel algorithm for fast inserts, deletes, and updates in B-trees, while providing the same level of read performance. B-epsilon trees trade space used for pivot keys in each node for space used to buffer writes, and the article by Bender et al. explains how the algorithm works, as well as proving it to be faster than B-trees for writes.

Singh et al. present Beam, part of a Microsoft project with a goal of collecting more useful information about certain events from the Internet of Things (IoT). While one type of sensor can provide potentially useful information, having an abstraction for multiple sensors can better answer a query such as "Is someone home?"

Andrea Spadaccini and Kavita Guliani continue the series of articles about the practices of System Resource Engineers (SREs) within Google. They explain how SRE teams handle on-call, one of the many vexing issues facing anyone who supports software services, in a way that has proven to work well and be fair to all participants.

Brendan Burns explains the Kubernetes (pronounced koo-ber-net-tees) project. While Docker has made containers into an easy-to-use packaging system, Kubernetes focuses on managing the services presented by applications running in containers. Kubernetes presents a single IP address for a group of containers, handles load balancing, keeps the configured number of services running, and handles scaling and upgrades.

Andy Seely has more tips for technical managers. In this column, Andy explains how time management is different for managers (compared to sysadmins and other technical staff), and provides advice from his own experience on how to best manage your time.

Dave Beazley's Python column explains some new syntax in Python 3.5. * and ** have been available for use in function arguments, where the function needs to be able to accept a variable number of arguments. Version 3.5 extends how this syntax works, including for specifying keyword-only arguments and conversion of arguments.

David Blank-Edelman explains how you can get Perl to work with WordPress. WordPress currently has a WP-API plugin that might become a standard part of WordPress, and David demonstrates how to get that plugin to work gracefully with the CRUST Web service.

Dave Josephsen wanted to be able to monitor the relative performance of some apps on different laptops. Dave shows how to install and use the Nagios Cross-Platform Agent for Linux and Apple systems.

Dan Geer discusses the denominator of risk: when we attempt to calculate risk, how best to choose the number of systems at risk. When comparing the number of unpatched exploits to the number of potential targets (the denominator), knowing the denominator can make a huge difference.

Robert Ferrell decides to redesign the Internet for better security, working as a non-network non-specialist.

Mark Lamourine has two book reviews this time, on *The Essential Turing* and *Drift into Failure*.

Peter Salus has written another in his series of columns on the history of USENIX, covering the change from having two Annual Tech conferences each year to having many more focused workshops and conferences. Salus also discusses the journal *Computer Systems*.

We conclude this issue with a portion of an interview conducted with Dan Geer in 2000, where he talks about why he became President of the USENIX Board of Directors. We included these statements because Geer explains both where USENIX was at this time (much larger) and his own remarkably insightful projections about the future he imagined 15 years ago.

Speaking of the future, I think we will continue to see both containers and VMs used on the same system. Whether unikernels will become as popular is still up in the air. Containers and VMs provide something familiar, and it is always easier for people to continue dealing with the familiar than to launch into the wilderness of the new. If support for unikernel-based applications continues to grow, these streamlined packages are likely to become just as popular.

Resources

[1] James Bottomley and Pavel Emelyanov, "Containers," *;login:*, vol. 39, no. 5, October 2014: https://www.usenix.org/publications/login/october-2014-vol-39-no-5/containers.

[2] Unikernels: http://wiki.xenproject.org/wiki/Unikernels.

[3] Linux system calls: http://man7.org/linux/man-pages/man2/syscalls.2.html, http://asm.sourceforge.net/syscall.html.

[4] Microsoft, Supported System Calls: https://technet.microsoft.com/en-us/library/Cc754234.aspx.

[5] John R. Douceur, Jeremy Elson, Jon Howell, and Jacob R. Lorch, Microsoft Research, "Leveraging Legacy Code to Deploy Desktop Applications on the Web": http://www.usenix.org/events/osdi08/tech/full_papers/douceur/douceur_html/index.html.

Letters to the Editor

Hello,

I received an issue of *;login:* magazine, "Sysadmin and Distributed Computing" (April 2015) while attending SouthEast LinuxFest (SELF) in June. I was very impressed with your publication and am now thoroughly disgusted with *Wired* magazine.

There was a mention of a Student Programs contact program, and I wanted to ask if you already have a rep on the Virginia Tech campus. If you have a rep, I would like to talk to them; if not, I would be glad to set up a Web site for USENIX info and library, which I can restrict to campus authorization.

I'll also be glad to forward USENIX info to our student Linux Users Group, VTLUG, and the Tech Support and/or Sys Admin campus groups.

Denton Yoder
Computer Systems Engineer
Biological Systems Engineering
Virginia Tech

Rik,

Thank you…USENIX is a great org and *;login:* a great mag. When it arrives, I know there will be an hour coming up shortly where I can put on the headphones, kick back, and read about people and ideas that relax and educate my poor tired computational soul. Good things by good people working for a better Net.

Thanks, and I promise to get on the stick and start submitting. Cyberville here is going 90 mph and just getting warmed up. Look forward to seeing folks out in my neck of the woods for WOOT and then for LISA.

Keep the faith…

Best,
Hal Martin
University of Maryland, Baltimore County

Do you have a USENIX Representative on your university or college campus? If not, USENIX is interested in having one!

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

A campus rep's responsibilities include:

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

In return for being our "eyes and ears" on campus, the Campus Representative receives access to the members-only areas of the USENIX Web site, free conference registration once a year (after one full year of service as a Campus Representative), and electronic conference proceedings for downloading onto your campus server so that all students, staff, and faculty have access.

To qualify as a campus representative, you must:

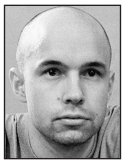
- Be full-time faculty or staff at a four-year accredited university
- Have been a dues-paying member of USENIX for at least one full year in the past

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

usenix THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

The Rise and Fall of the Operating System

ANTTI KANTEE



Antti has been an open source OS committer for over 15 years and believes that code which works in the real world is not born, it is made. He is a fan of simplest possible solutions. Antti lives in Munich and can often be seen down by the Isar River when serious thinking is required.
pooka@rumpkernel.org

An operating system is an arbitrary black box of overhead that enables well-behaving application programs to perform tasks that users are interested in. Why is there so much fuss about black boxes, and could we get things done with less?

Historical Perspective

Computers were expensive in the '50s and '60s. For example, the cost of the UNIVAC I in 1951 was just short of a million dollars [1]. Accounting for inflation, that is approximately nine million dollars in today's money. It is no wonder that personal computing had not been invented back then. Since it was desirable to keep millions of dollars of kit doing something besides idling, batch scheduling was used to feed new computations and keep idle time to a minimum.

As most of us intuitively know, reaching the solution of a problem is easier if you are allowed to stumble around with constant feedback, as compared to a situation where you must have holistic clairvoyance over the entire scenario before you even start. The lack of near-instant feedback was a problem with batch systems. You submitted a job, context switched to something else, came back the next day, context switched back to your computation, and discovered the proverbial missing comma in your program.

To address the feedback problem, time-sharing was invented. Users logged into a machine via a teletype and got the illusion of having the whole system to themselves. The time-sharing operating system juggled between users and programs. Thereby, poetic justice was administered: the computer was now the one context-switching, not the human. Going from running one program at a time to running multiple at the "same" time required more complex control infrastructure. The system had to deal with issues such as hauling programs in and out of memory depending on if they were running or not (swapping), scheduling the tasks according to some notion of fairness, and providing users with private, permanent storage (file systems). In other words, 50 years ago they had the key concepts of current operating systems figured out. What has happened since?

It's Called Hardware Because It Makes Everything Hard

When discussing operating systems, it is all but mandatory to digress to hardware, another black box. After all, presenting applications with a useful interface to hardware is one of the main tasks of an operating system, time-sharing or otherwise. So let's get that discussion out of the way first. The question is: why does hardware not inherently present a useful interface to itself? We have to peer into history.

I/O devices used to be simple, very simple. The intelligent bit of the system was the software running on the CPU. It is unlikely that manufacturers of yore desired to make I/O devices simpler than what they should be. The back-then available semiconductor technologies simply did not feasibly allow building complex I/O devices. An example of just how hopeless hardware used to be is the *rotational delay* parameter in old versions of the Berkeley Fast File System. That parameter controlled how far apart, rotationally speaking, blocks had to be written so that contiguous I/O could match the spinning of the disk. Over the years, adding more processing power to storage devices became feasible, and we saw many changes: fictional disk geometry, I/O buffering, non-spinning disks, automated bad block tracking, etc. As a result of the added processing power, approaches where the systems software pretends it still knows the internal details of devices, e.g., rotational delay, are obsolete or at least faltering.

As a result of added I/O device processing power, what else is obsolete in the software/hardware stack? One is tempted to argue that everything is obsolete. The whole hardware/software stack is bifurcated at a seemingly arbitrary position which made sense 30 years ago, but no longer. Your average modern I/O device has more computing power than most continents had 30 years ago. Pretending that it is the same dumb device that needs to be programmed by flipping registers with a sharpened toothpick results in sad programmers and, if not broken, at least suboptimal drivers. Does doing 802.11 really require 30k+ lines of driver code (including comments), 80k+ lines of generic 802.11 support, and a 1 MB firmware to be loaded onto the NIC? For comparison, the entire 4.3BSD kernel from 1986 including all device drivers, TCP/IP, the file system, system calls, and so forth is roughly 100k lines of code. How difficult can it be to join a network and send and receive packets? Could we make do with 1k lines of system-side code and 1.01 MB of firmware?

The solution for hardware device drivers is to push the complexity where it belongs in 2015, not where it belonged in 1965. Some say they would not trust hardware vendors to get complex software right, and therefore the complexity should remain in software running on the CPU. As long as systems software authors cannot get software right either, there is no huge difference in correctness. It is true that having most of the logic in an operating system does carry an advantage due to open source systems software actually being open source. Everyone who wants to review and adjust the 100k+ lines of code along their open source OS storage stack can actually do so, at least provided they have some years of spare time. In contrast, when hardware vendors claim to support "open source," the open source drivers communicate with an obfuscated representation of the hardware, sometimes through a standard interface such as SATA AHCI or HD audio, so in reality the drivers reveal little of what is going on in the hardware.

The trustworthiness of complex I/O devices would be improved if hardware vendors truly understood what "open source" means: publishing the most understandable representation, not just any scraps that can be run through a compiler. Vendors might prefer to not understand, especially if we keep buying their hardware anyway. Would smart but non-open hardware be a disaster? We can draw some inspiration from the automobile industry. Over the previous 30 years, we lost the ability to fix our cars and tinker with them. People like to complain about the loss of that ability. Nobody remembers to complain about how much better modern cars perform when they are working as expected.

Technology should encapsulate complexity and be optimized for the common case, not for the worst case, even if it means we, the software folk, give up the illusion of being in control of hardware.

If It Is Broken, Don't Not Fix It

The operating system is an old concept, but is it an outdated one? The early time-sharing systems isolated users from other users. The average general purpose operating system still does a decent job at isolating users from each other. However, that type of isolation does little good in a world that does not revolve around people logging into a time-sharing system from a teletype. The increasing problem is isolating the user from herself or himself.

Ages ago, when those who ran programs also wrote them, or at least had a physical interaction possibility with the people who did, you could be reasonably certain that a program you ran did not try to steal your credit card numbers. Also, back then your credit card information was not on the machine where you ran code, which may just as well be the root cause as to why nobody was able to steal it. These days, when you download a million lines of so-so trusted application code from the Internet, you have no idea of what happens when you run it on a traditional operating system.

The time-sharing system also isolates the system and hardware components from the unprivileged user. In this age when everyone has their own hardware—virtual if not physical—that isolation vector is of questionable value. It is no longer a catastrophe if an unprivileged process binds to transport layer ports less than 1024. Everyone should consider reading and writing the network medium as unlimited due to hardware no longer costing a million dollars, regardless of what an operating system does. The case for separate system and user software components is therefore no longer universal. Furthermore, the abstract interfaces that hide underlying power, especially that of modern I/O hardware, are insufficient for high-performance computing. If the interfaces were sufficient, projects looking at unleashing the hidden I/O power [3, 4] would not exist.

In other words, since the operating system does not protect the user from evil or provide powerful abstractions, it fails its mission in the modern world. Why do we keep on using such systems? Let us imagine the world of computing as a shape sorter. In the beginning, all holes were square: all computation was done on a million-dollar machine sitting inside of a mountain. Square pegs were devised to fit the holes. The advent of time-sharing brought better square pegs, but it did so in the confines of the old scenario of the mountain-machine. Then the world of computing diversified. We got personal computing, we got mobile devices, we got IoT, we got the cloud. Suddenly, we had round holes, triangular holes, and the occasional trapezoid and rhombus. Yet, we are still fascinated by square-shaped pegs, and desperately try to cram them into every hole, regardless of whether they fit.

Why are we so fascinated with square-shaped pegs? What happens if we throw away the entire operating system? The first problem with that approach is, and it is a literal show-stopper, that applications will fail to run. Already in the late 1940s computations used subroutine libraries [2]. The use of subroutine libraries has not diminished in the past 70 years, quite to the contrary. An incredible amount of application software keeping the Internet and the world running has been written against the POSIX-y interfaces offered by a selection of operating systems. No matter how much you do not need the obsolete features provided by the square peg operating system, you do want the applications to work. From-scratch implementations of the services provided by operating systems are far from trivial undertakings. Just implementing the 20-or-so flags for the `open()` call in a real-world-bug-compatible way is far from trivial.

Assuming you want to run an existing libc/application stack, you have to keep in mind that you still have roughly 199 system calls to go after `open()`. After you are done with the system calls, you then have to implement the actual components that the system calls act as an interface to: networking, file systems, device drivers, etc. After all that, you are finally able to get to the most time-consuming bit: testing your implementation in the real world and fixing it to work there. In essence, we are fascinated by square-shaped pegs because our applications rest on the support provided by those pegs. That is why we are stuck in a rut and few remember to look at the map.

There Is No Such Thing as Number One

The guitarist Roy Buchanan was confronted with a yell from the audience titling him as number one. Buchanan's response was: "There is no such thing as number one … but I love you for thinking about it, thank you very much." The response contains humble wisdom: no matter how good you are at some style(s), you can never be the arch master of all the arts. Similarly, in the ages past the mountain-machine, there is no one all-encompassing operating system because there are so many styles to computing. We need multiple solutions for multiple styles. The set presented below is not exhaustive but presents some variations from the mountain-machine style.

Starting from the simplest case, there is the embedded style case where you run one trust-domain on one piece of hardware. There, you simply need a set of subroutines (drivers) to enable your application to run. You do not need any code that allows the single-user, single-application system to act like a time-sharing system with multiple users. Notably, the single-application system is even simpler and more flexible than the single-user system [5], which, in turn, is simpler and more flexible than the multi-user system.

Second, we have the cloud. Running entire time-sharing systems as the provisioning unit on the cloud was not the ticket. As a bootstrap mechanism it was brilliant: everything worked like it worked without virtualization, so the learning curve could be approximated as having a zero-incline. In other aspects, the phrase "every problem in operating systems can be solved by *removing* layers of indirection" was appropriate. The backlash to the resource wastage of running full operating systems was containers, i.e., namespace virtualization provided by a single time-sharing kernel.

While containers are cheaper, the downside is the difficulty in making guarantees about security and isolation between guests. The current cloud trend is gearing towards *unikernels*, a term coined and popularized by the MirageOS project [6], where the idea is that you look at cloud guests just like you would look at single-application hardware. The hypervisor provides the necessary isolation and controls guest resource use. Since the hypervisor exposes only a simple hardware-like interface to the guest, it is much easier to reason about what can and should happen than it is to do so with containers. Also, the unikernel can be optimized for each application separately, so the model does not impose limiting abstractions either. Furthermore, if you can reasonably partition your computations so that one application instance requires at most one full-time core, most of the multicore programming performance problems simply disappear.

We also need to address the complex general purpose desktop/mobile case, which essentially means striking a balance between usability and limiting what untrusted applications can do. Virtualization would provide us with isolation between applications, but would it provide too much isolation?

Notably, when you virtualize, it is more difficult to optimize resource usage, since applications do not know how to play along in the grand ecosystem. For the cloud, that level of general ignorance is not a huge problem, since you can just add another datacenter to your cloud.

You cannot add another datacenter into your pocket in case your phone uses the local hardware resources in an exceedingly slack manner. Time will tell if virtualization adapted for the desktop [7] is a good enough solution, or if more fine-grained and precise methods [8] are required, or if they both are the correct answer given more specific preconditions. Even on the desktop, the square peg is not the correct shape: we know that the system will be used by a single person and that the system does not need to protect the user from non-existent other users. Instead, the system should protect the user from malware, spyware, trojans, and anything else that can crawl up the network pipe.

What We Are Doing to Improve Things

We can call them drivers, we can call them components, we can call them subroutines, we can call them libraries, but we need the pegs at the bottom of the computing stack for our applications to work. In fact, everything apart from the topmost layer of the software stack is a library. These days, with virtually unlimited hardware, it is mostly a matter of taste whether something is a "system driver" or "application library."

Rolling your own drivers is a hopeless battle. To address that market, we are providing componentized, reusable drivers at **http://rumpkernel.org/**. Those drivers come unmodified from a reputable kernel. Any approach requiring modification (aka porting) and maintenance induces an unbearable load for anything short of the largest projects with vast amounts of developer resources.

Treating the software stack as a ground-up construction of driver components gives the freedom to address each problem separately, instead of trying to invent ways to make the problem isomorphic to a mountain-machine. Drivers lifted from a time-sharing system will, of course, still exhibit time-sharing characteristics—there is no such thing as number one with drivers either. For example, the TCP/IP driver will still prevent non-root from binding to ports less than 1024. For example, in a unikernel, you are free to define what root or non-root means or simply compile the port check out of the driver. You can perform those modifications individually to suit the needs of each application. As a benefit, applications written for time-sharing-y, POSIX-y systems will not know what hit them. They will simply work because the drivers provide most everything that the applications expect.

We ended up building a unikernel based on the drivers offered by rump kernels via rumpkernel.org: *Rumprun*. We were not trying to build an OS-like layer but one day simply realized that we could build one which would just work, with minimal effort. The noteworthiness of the Rumprun unikernel does not come from the fact that existing software such as Nginx, PHP, and mpg123 can be cross-compiled in the normal fashion and then run directly on the cloud or on bare metal. The noteworthiness comes from the fact that the implementation is a few thousand lines of code … plus drivers. The ratio of drivers to "operating system" is on the order of 100:1, so there is very little *operating* system in there. The Rumprun implementation is that of an *orchestrating* system, which conducts the drivers.

Conclusion

Time-sharing systems were born over 50 years ago, a period from which we draw our concept of the operating system. Back then, hardware was simple, scarce, and sacred, and those attributes drove the development of the concepts of the system and the users. In the modern world, computing is done in a multitude of ways, and the case for the all-encompassing operating system has been watered down. Advances in semiconductor technology have enabled hardware to be smart, but hardware still exposes dumb interfaces, partially because we are afraid of smart hardware.

The most revered feature of the modern operating system is support for running existing applications. Minimally implemented application support is a few thousand lines of code plus the drivers, as we demonstrated with the Rumprun unikernel. Therefore, there is no reason to port and cram an operating system into every problem space. Instead, we can split the operating system into the "orchestrating system" (which also has the catchy OS acronym going for it) and the drivers. Both have separate roles. The drivers define what is possible. The orchestrating system defines how the drivers should work and, especially, how they are not allowed to work. The two paths should be investigated relatively independently as opposed to classic systems development where they are deeply intertwined.

References

[1] http://www.computerhistory.org/timeline/?category=cmptr.

[2] M. Campbell-Kelly, "Programming the EDSAC: Early Programming Activity at the University of Cambridge," *IEEE Annals of the History of Computing,* vol. 2, no. 1 (January–March 1980), pp. 7–36.

[3] S. Peter, J. Li, Irene Zhang, D. R. K. Ports, D. Woos, A. Krishnamurthy, T. Anderson, T. Roscoe, "Arrakis: The Operating System Is the Control Plane," *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation,* (2014), pp. 1–16.

[4] L. Rizzo, "netmap: A Novel Framework for Fast Packet I/O," *Proceedings of the USENIX Annual Technical Conference* (2012), pp. 101–112.

[5] B. Lampson and R. Sproull, "An Open Operating System for a Single-User Machine," *ACM Operating Systems Rev.,* vol. 11, no. 5 (Dec. 1979), pp. 98–105.

[6] MirageOS: https://mirage.io/.

[7] Qubes OS: https://www.qubes-os.org/.

[8] Genode Operating System Framework: http://genode.org/.

SYSTEMS

Trading Latency for Performance in Data-Intensive Applications

JACOB NELSON, BRANDON HOLT, BRANDON MYERS, PRESTON BRIGGS, SIMON KAHAN, LUIS CEZE, MARK OSKIN

Jacob Nelson received a PhD in computer science from the University of Washington in 2014. His research interests include computer architecture and runtime systems for big data and high-performance computing. nelson@cs.washington.edu

Brandon Holt is a PhD student in Computer Science and Engineering at the University of Washington, advised by Luis Ceze and Mark Oskin. He is interested in programming models, compilers, and systems for clusters, especially abstractions to mitigate real-world challenges like high contention. bholt@cs.washington.edu

Brandon Myers is a PhD candidate and Lecturer in Computer Science and Engineering at the University of Washington, advised by Bill Howe and Mark Oskin. He is interested in building systems to enable fast and flexible parallel programming, at the intersection of high performance computing, data management, and architecture. bdmyers@cs.washington.edu

Preston Briggs is a Senior Engineer at Reservoir Labs and an Affiliate Professor in Computer Science and Engineering at the University of Washington. He received a PhD in computer science from Rice University in 1992. preston@cs.washington.edu

The rising importance of data-intensive applications has fueled the growth of a plethora of distributed computing frameworks, including Hadoop, Spark, and GraphLab. We have developed a system called Grappa [1, 2] to aid programmers in developing new frameworks. Grappa provides a distributed shared memory abstraction to hide complexity from the programmer, and takes advantage of parallelism in the data to hide remote access latency and to trade latency for more performance. These techniques allow it to outperform existing frameworks by up to an order of magnitude.

Data-Intensive Applications on Distributed Shared Memory

Software distributed shared memory (DSM) systems provide shared memory abstractions for clusters. Historically, these systems performed poorly, largely due to limited inter-node bandwidth, high inter-node latency, and the design decision of piggybacking on the virtual memory system for seamless global memory accesses. Past software DSM systems were largely inspired by symmetric multiprocessors, attempting to scale that programming mindset to a cluster. However, applications were only suitable for them if they exhibited significant locality, limited sharing, and coarse-grained synchronization—a poor fit for many modern data-intensive applications.

DSM offers the promise of simpler implementations of data-intensive application frameworks. Figure 1 shows a minimal example of a "word count"-like application in actual Grappa DSM code. The input array, `chars`, and output hash table, `cells`, are distributed over multiple nodes. A parallel loop runs on all nodes to process shards of the input array, hashing each key to its cell and incrementing the corresponding count atomically. The code looks similar to plain shared-memory code, yet it spans multiple nodes and scales efficiently.

Applying the DSM concept to common data-intensive computing frameworks is similarly straightforward:

MapReduce. Data parallel operations like Map and Reduce are simple to think of in terms of shared memory. Map is simply a parallel loop over the input (an array or other distributed data structure). It produces intermediate results into a hash table similar to that in Figure 1. Reduce is a parallel loop over all the keys in the hash table.

Vertex-centric. GraphLab/PowerGraph is an example of a vertex-centric execution model, designed for implementing machine-learning and graph-based applications. Its three-phase gather-apply-scatter (GAS) API for vertex programs enables several optimizations pertinent to natural graphs. Such graphs are difficult to partition well, so algorithms traversing them exhibit poor locality. Each phase can be implemented as a parallel loop over vertices, but fetching each vertex's neighbors results in many fine-grained data requests.

Relational query execution. Decision support, often in the form of relational queries, is an important domain of data-intensive workloads. All data is kept in hash tables stored in a DSM. Communication comes from inserting into and looking up in hash tables. One parallel loop builds a hash table, followed by a second parallel loop that filters and probes the hash

Luis Ceze is an Associate Professor of Computer Science and Engineering at the University of Washington. His research focuses on improving programmability, reliability, and energy efficiency of multiprocessor and multicore systems. luisceze@cs.washington.edu

Simon Kahan is an Affiliate Professor of Computer Science and Engineering at the University of Washington. His current research focuses on accelerating large-scale biological simulation and numerical linear algebra. skahan@cs.washington.edu

Mark Oskin is an Associate Professor of Computer Science and Engineering at the University of Washington. oskin@cs.washington.edu

```
// distributed input array
GlobalAddress<char> chars = load_input();

// distributed hash table:
using Cell = std::map<char,int>;
GlobalAddress<Cell> cells = global_alloc<Cell>(ncells);

forall(chars, nchars, [=](char& c) {
  // hash the char to determine destination
  size_t idx = hash(c) % ncells;
  delegate(&cells[idx], [=](Cell& cell)
  { // runs atomically
    if (cell.count(c) == 0) cell[c] = 1;
    else cell[c] += 1;
  });
});
```

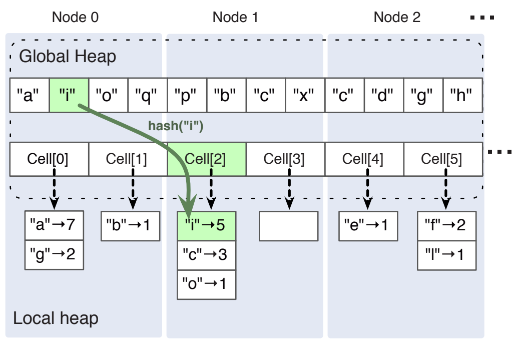

Node 0
Node 1
Node 2
Global Heap
"a" "i" "o" "q" "p" "b" "c" "x" "c" "d" "g" "h"
hash("i")
Cell[0] Cell[1] Cell[2] Cell[3] Cell[4] Cell[5]
"a"→7
"g"→2
"b"→1
"i"→5
"c"→3
"o"→1
"e"→1
"f"→2
"l"→1
Local heap

Figure 1: "Character count" with a simple hash table implemented using Grappa's distributed shared memory

table, producing the results. These steps rely heavily on consistent, fine-grained updates to hash tables.

While these frameworks are easy to express conceptually in a DSM system, obtaining good performance can be challenging for a number of reasons:

Small messages. Programs written to a shared memory model tend to access small pieces of data. On a DSM system this requires communication. What were simple load or store operations become implicit, complex transactions involving the network. When these messages are small (~32 bytes), the network (optimized for multi-kilobyte packets) struggles to achieve a fraction of its peak throughput.

Poor locality. Data-intensive applications often exhibit poor locality. For example, the volume of communication in GraphLab's gather and scatter operations is a function of the graph partition. Complex graphs frustrate even the most advanced partitioning schemes. This leads to poor spatial locality. Moreover, which vertices are accessed varies from iteration to iteration. This leads to poor temporal locality.

Need for fine-grained synchronization. Typical data-parallel applications offer coarse-grained concurrency with infrequent synchronization—e.g., between phases of processing a large chunk of data. Conversely, graph-parallel applications exhibit fine-grained concurrency with frequent synchronization—e.g., when done processing work associated with a single vertex. Therefore, for a DSM solution to be general, it needs to support fine-grained synchronization efficiently.

Fortunately, data-intensive applications have properties that can be exploited to make DSMs efficient: their abundant data parallelism enables high degrees of concurrency; and their performance depends not on the *latency* of execution of any specific parallel task, as it would in, for example, a Web server, but rather on the aggregate execution time (i.e., *throughput*) of *all* tasks.

Grappa Design

Figure 2 shows an overview of Grappa's DSM system. We will first describe the multithreading and communication layers and then explore the distributed shared memory layer, which is built on top of these lower-level components. Our recent USENIX ATC paper [2] describes these in more detail.

Expressing and Exploiting Parallelism

Work is most commonly expressed in Grappa using parallel `for` loops. Tasks may also be spawned individually, with optional data locality constraints. Under the hood, both methods

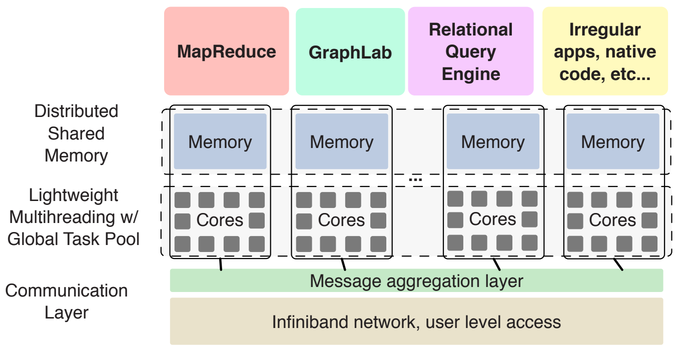

MapReduce
GraphLab
Relational Query Engine
Irregular apps, native code, etc...
Distributed Shared Memory
Memory
Memory
Memory
Memory
Lightweight Multihreading w/ Global Task Pool
Cores
Cores
Cores
Cores
Communication Layer
Message aggregation layer
Infiniband network, user level access

Figure 2: Grappa's distributed shared memory abstraction is designed to make it easy to implement data-intensive application frameworks. It uses lightweight threads to tolerate remote access latencies by exploiting fine-grained parallelism in the data, and it transparently aggregates small messages into larger ones to improve communication performance.

work by pushing closures into a global task pool. These closures are generally expressed using C++11 lambda constructs to provide both code to execute and initial state. Tasks are executed by idle threads on cores across the system, which pull from the global task queue subject to the tasks' locality constraints. When a task executes a long-latency operation, it is suspended until the operation is complete; the core it is running on is kept busy with other, independent, work.

Grappa is built around a user level, cooperative multithreading system. Due to the large inter-node latencies that must be tolerated in a distributed system like Grappa, the scheduler is built to support on the order of 1000 concurrent threads per core. We do this by storing and switching minimal context for threads, and by prefetching thread contexts into cache before switching to them, thereby enabling context switches to happen at a rate limited only by DRAM bandwidth, rather than cache miss latency.

Communication Support

Grappa's communication layer has two components. The upper (user-level) layer is designed to support sending very small messages—tens of bytes—at a high rate, with low memory overhead. We use an asynchronous active message approach: the sender creates a message holding a C++11 lambda or other closure, and the receiver executes the closure. We take advantage of the fact that our homogeneous cluster hardware runs the same binary in every process: each message consists of a template-generated deserializer pointer, a byte-for-byte copy of the closure, and an optional dynamically sized data payload.

At the lower (network) level, Grappa moves these small messages over the network efficiently by transparently aggregating independent messages destined for common network destinations. This process, shown in Figure 3, works as follows. When a compute task sends a message, the data is not immediately placed on the network but instead is stored in a per-core buffer. A communication task runs periodically; when it finds a large group of messages headed for the same node, or messages that have been waiting for a long time, it serializes them into a single, large network packet, which it sends to the destination node. When the remote node receives the packet, it distributes the messages to their destination cores, where messages are deserialized and their handlers are executed.

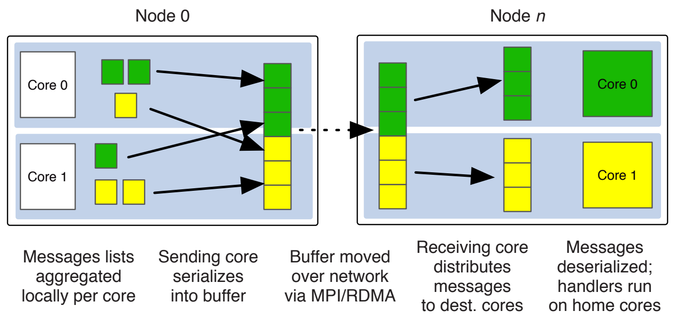

Node 0
Node n
Core 0
Core 1
Core 0
Core 1
Messages lists aggregated locally per core
Sending core serializes into buffer
Buffer moved over network via MPI/RDMA
Receiving core distributes messages to dest. cores
Messages deserialized; handlers run on home cores

Figure 3: Grappa achieves high throughput for small messages by automatically batching messages with a common destination in order to move larger packets over the network, amortizing network invocation and delivery costs over multiple messages.

Grappa uses RDMA to move messages, but only indirectly. User-level messages are created using non-temporal memory operations and prefetches to avoid cache pollution. Aggregated messages are moved between nodes using MPI for portability, tuned to use RDMA when available. By amortizing network invocation costs across many messages, we are able to obtain significantly better performance than using native RDMA operations: on a simple random-access benchmark, Grappa's DSM operations performed atomic increments 25 times faster than native RDMA increments on our 128-node AMD Interlagos cluster connected with 40 Gb Mellanox ConnectX-2 InfiniBand cards.

Addressing in Grappa's Distributed Shared Memory

In Grappa, memory is partitioned across cores; each byte is considered local to a single core within a node in the system. Accesses to local memory occur through conventional pointers. Local pointers cannot refer to memory on other cores; they are valid only on their home core. Local accesses are used to reference many things in Grappa, including the stack associated with a task, scheduling and debugging data structures, and the slice of global memory local to a core.

Accesses to non-local memory occur through global pointers. Grappa allows any local data on a core's stacks or heap to be exported to the global address space and made accessible to other cores across the system. This uses a partitioned global address space (PGAS) model, where each address is a tuple of a core ID and an address local to that core.

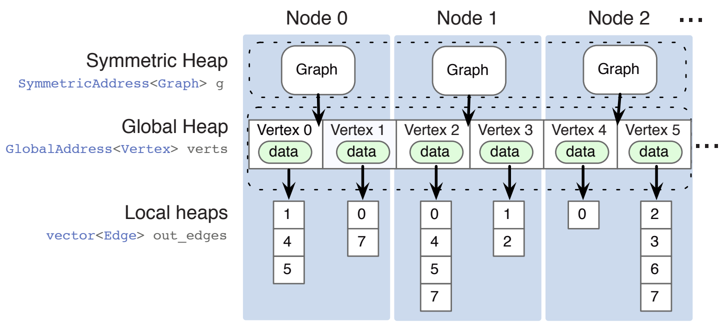


Figure 4: Using global addressing for graph layout

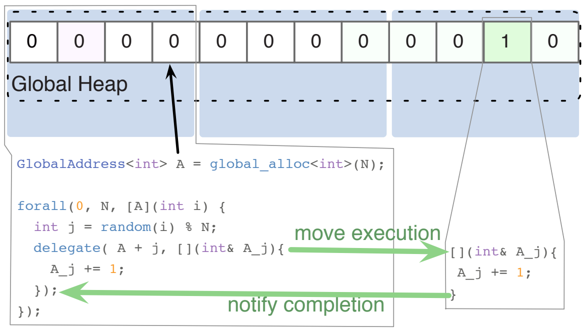
Grappa also supports *symmetric* allocations, which reserves space for a copy of an object on every core in the system. The behavior is identical to performing a local allocation on all cores, but the local addresses of all the allocations are guaranteed to be identical. Symmetric objects are often treated as a proxy for a global object, holding local copies of constant data, or allowing operations to be transparently buffered. A separate publication [3] describes how this was used to implement Grappa's synchronized global data structures, including vector and hash map.

Figure 4 shows an example of how global, local, and symmetric memory can all be used together for a simple graph data structure. In this example, vertices are allocated from the global heap, automatically distributing them across nodes. Symmetric pointers are used to access local objects which hold information about the graph, such as the base pointer to the vertices, from any core without communication. Finally, each vertex holds a vector of edges allocated from their core's local heap, which other cores can access by going through the vertex.

Accessing Memory with Delegate Operations

Access to Grappa's distributed shared memory is provided through *delegate* operations, which are short operations performed at a memory location's home core. When the data access pattern has low locality, it is more efficient to modify the data on its home core rather than bringing a copy to the requesting core and returning a modified version. While delegates can trivially implement *read/write* operations to global memory, they can also implement more complex *read-modify-write* and synchronization operations (e.g., *fetch-and-add*, mutex acquire, queue insert).

We have explored two approaches for expressing delegate operations. In the first, the programmer calls functions in Grappa's API—a change from the traditional DSM model. Generally, these delegates are expressed as C++11 lambdas or other closures; Figure 5 shows an example. The second approach uses a compiler pass implemented with LLVM to automatically identify and extract productive delegate operations from ordinary code; this approach is explored in another publication [4]. In practice, we usually use the library-based approach, since exploiting available locality is important for getting maximum performance in a distributed system, and writing explicit delegate operations is an easy way to express that locality.



```
GlobalAddress<int> A = global_alloc<int>(N);

forall(0, N, [A](int i) {
  int j = random(i) % N;
  delegate( A + j, [](int& A_j){
    A_j += 1;
  });
});
```

```
[](int& A_j){
  A_j += 1;
}
```

Figure 5: Grappa delegate example

Delegates and Memory Consistency

Memory consistency and efficient synchronization are a result of delegation in Grappa.

All sharing, whether between cores within a node or between two nodes, as well as synchronization, is done via delegate operations. A delegate operation can execute arbitrary code subject to two restrictions: first, the code can reference only data local to the core on which the delegate is executing; and second, the code may not execute operations that lead to a context switch.

Since delegate operations execute on a particular core in some serial order and only touch data owned by that core, they are guaranteed to be globally linearizable, with their updates visible to all cores across the system in the same order. In addition, only one synchronous delegate will be in flight at a time from a particular task, so synchronization operations from a particular task are not subject to reordering. Moreover, once one core is able to see an update from a synchronous delegate, all other cores are too. Consequently, all synchronization operations execute in program order and are made visible in the same order to all cores in the system. These properties are sufficient to guarantee a memory model that offers sequential consistency for data-race-free programs, which is what underpins C/C++.

The synchronous property of delegates provides a clean model but can be overly restrictive for operations that are protected by collective synchronization like a global barrier. For such cases, we also support *asynchronous delegates*, which, like delegate operations, execute non-blocking regions of code atomically on a single core's memory. Asynchronous delegates are treated as task spawns in the memory model and are generally linked with a collective synchronization operation to detect completion.

Measuring Performance with Prototype Application Frameworks

We implemented three prototype application frameworks in Grappa. The first is an in-memory MapReduce implementation, which we compared with Spark [5] with fault tolerance disabled. The second is a distributed backend for the Raco relational algebra compiler and optimization framework [6], which we compared with Shark [7]. The third is a vertex-centric programming framework in the spirit of GraphLab [8], which we compare with native GraphLab.

The full performance results are reported in our USENIX ATC paper [2]; here we provide a brief summary. On the cluster mentioned previously, we found the Grappa MapReduce implementation to be 10 times faster than Spark on a k-means clustering benchmark. The Grappa query processing engine was 12.5 times faster than Shark on the SP2Bench benchmark suite [9]. The Grappa vertex-centric framework was 1.33 times faster than GraphLab on graph analytics benchmarks from the GraphBench suite [10].

Conclusion

Our work builds on the premise that writing data-intensive applications and frameworks in a shared memory environment is simpler than developing custom infrastructure from scratch. Based on this premise, we show that a DSM system can be efficient for this application space by judiciously exploiting the key application characteristics of concurrency and latency tolerance. Our work demonstrates that frameworks such as MapReduce, vertex-centric computation, and query execution can be easy to build and are efficient in a DSM system.

Acknowledgments

This work was supported by NSF Grant CCF-1335466, Pacific Northwest National Laboratory and gifts from NetApp and Oracle.

References

[1] Grappa Web site and source code: http://grappa.io/.

[2] Jacob Nelson, Brandon Holt, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, and Mark Oskin, “Latency-Tolerant Software Distributed Shared Memory,” in *Proceedings of the 2015 USENIX Annual Technical Conference (USENIX ATC ’15),* Santa Clara, CA.

[3] Brandon Holt, Jacob Nelson, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, and Mark Oskin, “Flat Combining Synchronized Global Data Structures,” International Conference on PGAS Programming Models (PGAS), October 2013.

[4] Brandon Holt, Preston Briggs, Luis Ceze, and Mark Oskin, “Alembic: Automatic Locality Extraction via Migration,” in *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA ’14),* 2014.

[5] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica, “Spark: Cluster Computing with Working Sets,” in *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing (HotCloud ’10),* 2010.

[6] Raco: The relational algebra compiler: https://github.com/uwescience/datalogcompiler, April 2014.

[7] Reynold S. Xin, Josh Rosen, Matei Zaharia, Michael J. Franklin, Scott Shenker, and Ion Stoica, “Shark: SQL and Rich Analytics at Scale,” in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD ’13),* 2013.

[8] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin, “PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs,” in *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI ’12),* 2012.

[9] Michael Schmidt, Thomas Hornung, Georg Lausen, and Christoph Pinkel, “SP2Bench: A SPARQL Performance Benchmark,” *Computing Research Repository,* abs/0806.4627, 2008.

[10] GraphBench: http://graphbench.org/, 2014.

Thread and Memory Placement on NUMA Systems
Asymmetry Matters

BAPTISTE LEPERS, VIVIEN QUÉMA, AND ALEXANDRA FEDOROVA

Baptiste Lepers is a postdoc at Simon Fraser University. His research topics include performance profiling, optimizations for NUMA systems, and multicore programming. He likes to spend his weekends in the mountains, hiking and biking. baptiste.lepers@gmail.com

Vivien Quéma is a Professor at Grenoble INP (ENSIMAG). His research is about understanding, designing, and building (distributed) systems. He works on Byzantine fault tolerance, multicore systems, and P2P systems. vivien.quema@grenoble-inp.fr

Alexandra Fedorova is an Associate Professor at the University of British Columbia. Her research focuses on building systems software that facilitates synergy between applications and hardware. In her spare time, she consults for MongoDB. sasha@ece.ubc.ca

Industry uses NUMA multicore machines for its servers. On NUMA machines, the conventional wisdom is to place threads close to the memory they access, and to collocate the threads that share data on the same CPU nodes. However, this is often not optimal. Indeed, modern NUMA machines have asymmetric interconnect links between CPU nodes, which can strongly affect performance, with best placement outperforming worst placement on nodes by a factor of almost two. We present the *AsymSched* algorithm, which uses CPU performance counters to measure performance and dynamically migrate threads and memory to achieve the best placement.

Modern Computers Are Asymmetric

Modern multicore machines are structured as several CPU/memory nodes connected via an interconnect. These architectures are usually characterized by non-uniform memory access times (NUMA), meaning that the latency of data access depends on *where* (which CPU-cache or memory node) the data is located. For this reason, the placement of threads and memory plays a crucial role in performance. To that end, both researchers and practitioners designed a variety of NUMA-aware thread and memory placement algorithms [8, 7, 5, 13, 14, 4]. Their insight is to place threads close to their memory, to spread the memory pages across the system to avoid the overload on memory controllers and interconnect links, to collocate data-sharing threads on the same node while avoiding memory controller contention, and to segregate threads competing for cache and memory bandwidth on different nodes. These algorithms assume that the interconnect between nodes is symmetric: given any pair of nodes connected via a direct link, the links have the same bandwidth and the same latency. *On modern NUMA systems this is not the case.*

Figure 1 depicts an AMD Bulldozer NUMA machine with eight nodes, each hosting eight cores. Interconnect links exhibit many disparities:

1. Links have different bandwidths: some have 16-bit width, some have 8-bit width.
2. Some links can send data faster in one direction than in the other (i.e., one side sends data at 3/4 the speed of a 16-bit link, while the other side can only send data at the speed of an 8-bit link). We call these links 16/8-bit links.
3. Links are shared differently. For instance, the link between nodes 4 and 3 is only used by these two nodes, while the link between nodes 2 and 3 is shared by nodes 0, 1, 2, 3, 6, and 7.
4. Some links are unidirectional. For instance, node 7 sends requests directly to node 3, but node 3 routes its answers via node 2. This creates an asymmetry in read/write bandwidth: node 7 can write at 4 GB/s to node 3, but can only read at 2 GB/s.

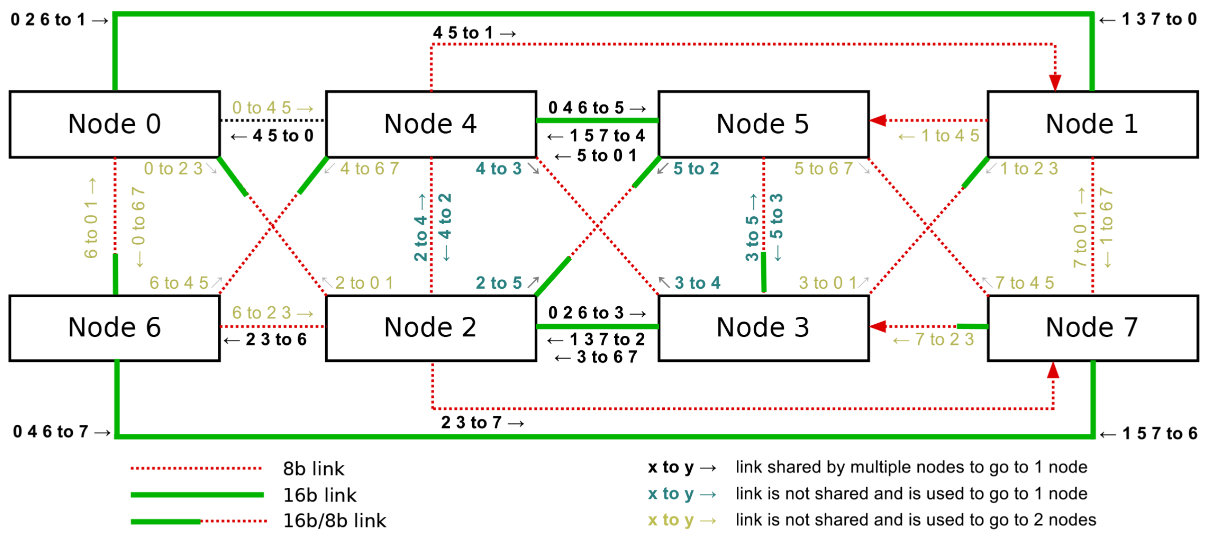


Figure 1: Modern NUMA systems, with eight nodes. The width of links varies; some paths are unidirectional (e.g., between 7 and 3), and links may be shared by multiple nodes. Machine A has 64 cores (8 cores per node—not represented in the picture), and machine B has 48 cores (6 cores per node). Not shown in the picture: the links between nodes 4 and 1 and between nodes 2 and 7 are bidirectional on machine B. This changes the routing of requests from node 7 to 2 and node 1 to 4.

Impact of Asymmetry on Performance

The asymmetry of interconnect links has dramatic and at times surprising effects on performance. Figure 2 shows the performance of 20 different applications on the 64-core machine shown in Figure 1. Each application runs with 24 threads, so it needs three nodes to run on. We vary *which* three nodes are assigned to the application and hence the *connectivity* between the nodes. The relative placement of threads and memory on those nodes is *identical* in all configurations. The only difference is *how the chosen nodes are connected*. The figure shows the performance on the best-performing and the worst-performing subset of nodes for that application compared to the average (obtained by measuring the performance on all 336 unique subsets of nodes and computing the mean). We make several observations. First, the performance on the best subset is up to 88% faster than the average, and the performance on the worst subset is up to 44% slower. Second, the maximum performance difference between the best and the worst subsets is 237% (for FaceRec). Finally, the mean difference between the best and worst subsets is 40% and the median 14%.

We measured that the memory accesses performed by FaceRec are approximately 600 cycles faster when running on the best subset of nodes relative to the average, and 1400 cycles faster relative to the worst. The latency differences are tightly correlated with the performance difference between configurations.

To further understand the cause of very high latencies on “bad” configurations, we analyzed streamcluster, an application from

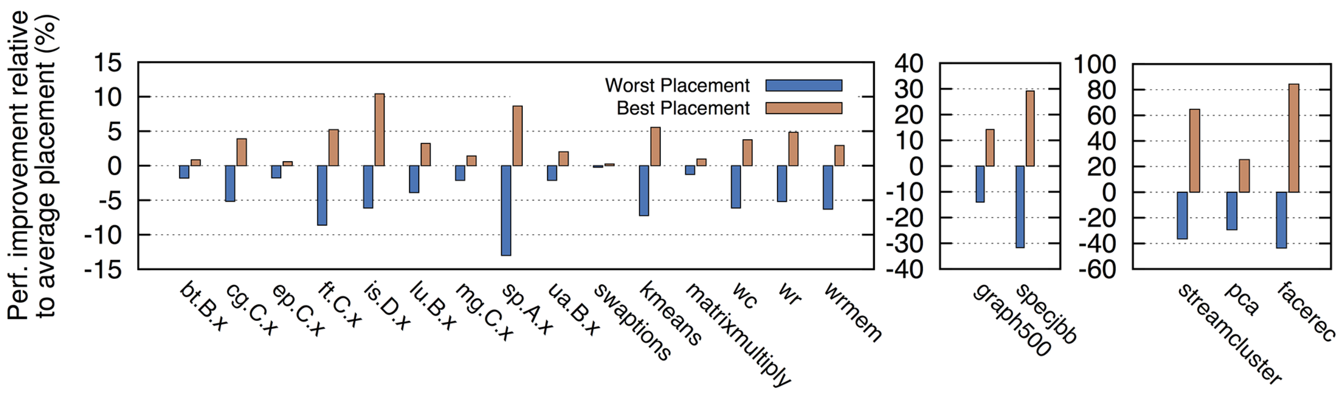
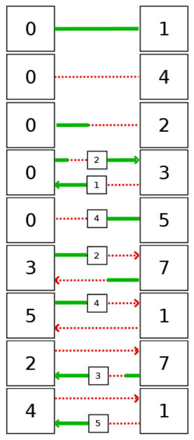


Figure 2: Performance difference between the best, and worst thread placement with respect to the average thread placement on Machine A. Applications run with 24 threads on three nodes. Graph500, SPECjbb, streamcluster, PCA, and FaceRec are highly affected by the choice of nodes and are shown separately with a different y-axis range.



Master thread node	Execution time (s)	Diff with 0-1 (%)	Latency of memory accesses (cycles) (compared to 0-1(%))	% accesses via 2-hop links	Bandwidth to the "master" node (MB/s)
-	148	0%	750	0	5598
-	228	56%	1169 (56%)	0	2999
0	228	56%	1179 (57%)	0	2973
2	168	15%	855 (14%)	0	4329
0	340	133%	1527 (104%)	98	1915
3	**185**	27%	1040 (39%)	98	3741
0	340	133%	1601 (113%)	98	1903
5	**228**	56%	1206 (61%)	98	2884
3	185	27%	1020 (36%)	0	3748
7	338	132%	1614 (115%)	98	1928
1	338	132%	1612 (115%)	98	1891
5	230	58%	1200 (60%)	0	2880
2	**167**	15%	867 (16%)	98	3748
7	225	54%	1220 (63%)	0	3014
4	230	58%	1205 (60%)	0	2959
1	**226**	55%	1203 (60%)	98	2880

Table 1: Performance of streamcluster executing with 16 threads on two nodes on machine A. The performance depends on the connectivity between the nodes on which streamcluster is executing and on the node on which the master thread is executing. Numbers in bold indicate two-hop configurations that are as fast or faster than some one-hop configurations.

the PARSEC [11] benchmark suite, which is among the most affected by the placement of its threads and memory. We ran streamcluster with 16 threads on two nodes. Table 1 presents the salient metrics for each possible two-node subset. Depending on which two nodes we chose, we observe large (up to 133%) disparities in performance. The data in Table 1 leads to several crucial observations:

- Performance is correlated with the latency of memory accesses.
- Surprisingly, the latency of memory accesses is not correlated with the number of hops between the nodes: some two-hop configurations (shown in bold) are faster than one-hop configurations.
- The latency of memory accesses is actually correlated with the *bandwidth* between the nodes. Note that this makes sense: the difference between one-hop vs. two-hop latency is only 80 cycles when the interconnect is nearly idle. So a higher number of hops alone cannot explain the latency differences of thousands of cycles.

As a summary, we can say that **bandwidth between the nodes matters more than the distance between them**.

Computers Are Increasingly Asymmetric

Asymmetric interconnect is not a new phenomenon. Nevertheless, its effects on performance are increasing as machines are built with more nodes and cores. We measured the performance of streamcluster on four different asymmetric machines: two recent machines with 64 and 48 cores, respectively, and eight nodes (Machines A and B, Figure 1), and two older machines with 24 and 16 cores, respectively, and four nodes (Machines C and D, not depicted). All of these machines use AMD Opteron processors. Machines A and B have highly asymmetric interconnect. Machines C and D have a less pronounced asymmetry. Machine C has full connectivity, but two of the links are slower than the rest. Machine D has links with equal bandwidth, but two nodes do not have a link between them.

Table 2 shows the performance of streamcluster with 16 threads on the best-performing and the worst-performing set of nodes on each machine. The performance difference between the best and worst configurations increases with the number of cores in the machine: from 3% for the 16-core machine to 133% for the 64-core machine. We explain this as follows:

1. On the 16-core Machine D, the only difference between configurations is the longer wire delay between the nodes that are not connected via a direct link. This delay is not significant compared to the extra latency induced by bandwidth contention on the interconnect.
2. The CPUs on 24-core Machine C have a low frequency compared to the other machines. As a result, the impact of longer memory latency is not as pronounced. More importantly, the network on this machine is still a fully connected mesh, so there is less asymmetry than on Machines A and B.
3. The 48- and 64-core Machines B and A offer a wider range of bandwidth configurations, which increases the difference between the best and the worst placements. The 64-core machine is more affected than the 48-core machine because it has more cores per node, which increases the effects of bandwidth contention.

Intel machines are currently built using symmetric interconnect links, but we believe that, as the number of nodes in systems increases, this will no longer remain true in the future.

Machine	Best Time	Worst Time	Difference
A (64 cores)	148s	340s	133%
B (48 cores)	149s	277s	85%
C (24 cores)	171s	229s	33%
D (16 cores)	255s	262s	3%

Table 2: Performance of streamcluster executing on two nodes on machine A, B, C, and D. The performance of streamcluster depends on the placement of its threads. The impact of thread placement is more important on recent machines (A and B) than on older ones (C and D).

Handling Asymmetry: The Challenges

To take into account interconnect asymmetry, the operating system should choose a "good" subset of nodes for each application. More precisely, the operating system should try, for each application, to **place threads and memory pages on a well-connected subset of nodes**. When an application executes on only two nodes on a machine similar to the one used in Table 1, the placement on the nodes connected with the widest (16-bit) link is always the best because it maximizes the bandwidth and minimizes the latency between the nodes. However, when an application needs more than two nodes to run, no configuration exists with 16-bit links between every pair of nodes, so the operating system must decide which nodes to pick. Besides, when there is more than one application running, the operating system needs to decide how to allocate the nodes among multiple applications. Designing such a thread and memory placement algorithm raises several challenges that we list below.

Nodes	% Perf. Relative to Best Subset	
	Streamcluster	SPECjbb
0, 1, 3, 4, and 7	-64%	0% (best)
2, 3, 4, 5, and 6	0% (best)	-9.4%

Table 3: Performance of streamcluster and SPECjbb on two different set of nodes on machine A, relative to the best set of nodes for the respective application

Efficient online measurement of communication patterns is challenging: The algorithm must measure the volume of CPU-to-CPU and CPU-to-memory communication for different threads in order to determine the best placement. This measurement process must be very efficient, because it must be done continuously in order to adapt to phase changes.

Changing the placement of threads and memory may incur high overhead: Frequent migration of threads may be costly, because of the associated CPU overhead, but most importantly because cache affinity is not preserved. Moreover, when threads are migrated to "better" nodes, it might be necessary to migrate their memory in order to avoid the overhead of remote accesses and overloaded memory controllers. Migrating large amounts of memory can be extremely costly. Thus, thread migration must be done in a way that minimizes memory migration.

Accommodating multiple applications simultaneously is challenging: Applications have different communication patterns and are thus differently impacted by the connectivity between the nodes they run on. As an illustration, Table 3 presents the performance of streamcluster and SPECjbb executing on two different sets of five nodes (the best set of nodes for the two applications, respectively). The two applications behave differently on these two sets of nodes: streamcluster is 64% slower on the best set of nodes for SPECjbb than on its own best set. The algorithm must, therefore, determine the best set of nodes for every application. Furthermore, it cannot always place each application on its best set of nodes, because applications may have conflicting preferences.

Selecting the best placement is combinatorially difficult: The number of possible application placements on an eight-node machine is very large (e.g., 5040 possible configurations for four applications executing on two nodes). So, (1) it is not possible to try all configurations *online* by migrating threads and then choosing the best configurations, and (2) doing even the simplest computation involving "all possible placements" can still add a significant overhead to a placement algorithm.

The AsymSched Algorithm

We designed *AsymSched* [9], a thread and memory placement algorithm that takes into account the bandwidth asymmetry of asymmetric NUMA systems. *AsymSched*'s goal is to maximize the bandwidth for *CPU-to-CPU communication*, which occurs between threads that exchange data, and *CPU-to-memory communication*, which occurs between a CPU and a memory node upon a cache miss. To that end, *AsymSched* places threads that perform extensive communication on relatively well-connected nodes, and places the frequently accessed memory pages such that the data requests are either local or travel across high-bandwidth paths.

AsymSched is implemented as a user-level process and interacts with the kernel and the hardware using system calls and `/proc` file system, but could also be easily integrated with the kernel scheduler if needed.

AsymSched relies on three main techniques to manage threads and memory:

1. **Thread migration**: changing the node where a thread is running
2. **Full memory migration**: migrating all pages of an application from one node to another
3. **Dynamic memory migration**: migrating only the pages that an application actively accesses as done in [7]

The operating principle of *AsymSched* is the following: *AsymSched* continuously gathers hardware counter values on the number of memory requests. Every second, *AsymSched* takes a thread placement decision. Roughly speaking, it groups threads of the same application that share data in virtual weighted *clusters*. The weight of a cluster represents the intensity of memory accesses performed between threads belonging to the cluster. Then *AsymSched* computes possible *placements* for all the clusters. A placement is an array mapping clusters to nodes. For each placement, *AsymSched* computes the maximum bandwidth that each cluster would receive if it were put in this placement. *AsymSched* selects the placement, ensuring that clusters with the highest weights will be scheduled on the nodes with the best connectivity. Finally, *AsymSched* estimates the overhead of memory migration induced by the new placement. If the overhead is deemed too high, the new placement will not be applied. Otherwise, *AsymSched* performs thread and memory migration to apply the new placement.

AsymSched implements two main optimizations. The first optimization allows **fast memory migrations**. When *AsymSched* performs full memory migration, all the pages located on one node are migrated to another node. The applications we tested have large working sets (up to 15 GB per node), and migrating pages is costly. Migrating 10 GB of data using the standard `migrate_pages` system call takes 51 seconds on average, making the migration of large applications impractical.

We therefore designed a new system call for memory migration. This system call performs memory migration without locks in most cases, and exploits the parallelism available on multicore machines. Using our system call, migrating memory between two nodes is on average 17x faster than using the default Linux system call and is only limited by the bandwidth available on interconnect links. Unlike the Linux system call, our system call can migrate memory from multiple nodes simultaneously. So if we are migrating the memory simultaneously between two pairs of nodes that do not use the same interconnect path, our system call will run about 34x faster.

The second optimization **avoids evaluating all possible placements.** It is based on two observations:

1. A lot of thread placement configurations are "obviously" bad. For instance, when a communication-intensive application uses two nodes, we only consider configurations with nodes connected with a 16-bit link.
2. Several configurations are equivalent (e.g., in the machine depicted in Figure 1, the bandwidth between nodes 0 and 1 and between nodes 2 and 3 is the same). To avoid estimating the bandwidth of *all* placements, we create a hash for each placement. The hash is computed so that equivalent configurations have the same hash.

Using simple dynamic programming techniques, we only perform computations on non-equivalent configurations. Our optimization allows skipping between 67% and 99% of computations in all tested configurations with clusters of two, three, or five nodes (e.g., with four clusters of two nodes, we only evaluate 20 configurations out of 5040).

AsymSched Assessment

We evaluated the performance achieved when using *AsymSched* on Machine A. The latter is equipped with four AMD Opteron 6272 processors, each with two NUMA nodes and eight cores per node (64 cores in total). The machine has 256 GB of RAM, uses HyperTransport 3.0, and runs Linux 3.9. We used several benchmark suites: the NAS Parallel Benchmarks suite [3], which is composed of numeric kernels; MapReduce benchmarks from Metis [10]; parallel applications from PARSEC [11]; Graph500 [1], a graph processing application with a problem size of 21; FaceRec from the ALPBench benchmark suite [6]; and SPECjbb [2] running on OpenJDK7.

Our goal was to evaluate the impact of asymmetry-aware thread placement *in isolation* from other effects, such as those stemming purely from collocating threads that share data on the same node. Performance benefits of sharing-aware thread clustering are well known [13]. *AsymSched* clusters threads that share data; the Linux thread scheduler, however, does not. We experimentally observed that Linux performed worse than clustered configurations. For instance, when Graph500 and SPECjbb are scheduled simultaneously, both run 23% slower on Linux than on an average clustered placement.

Since comparing Linux to *AsymSched* would not be meaningful because of that, we instead compare *AsymSched* to the best and the worst static placements of data-sharing thread clusters. When running *AsymSched*, thread clusters are initially placed on a randomly chosen set of nodes. We also compare the average performance achieved under all static placements that are unique in terms of connectivity. We obtain all unique static placements with respect to connectivity by examining the topology of the machine. There are 336 placements for single-application scenarios and 560 placements for multi-application scenarios.

Further, we want to isolate the effects of thread placement with *AsymSched* from the effects of dynamic memory migration. To that end, we compare *AsymSched* to the subset of our algorithm that performs the dynamic placement of memory only, turning off the parts performing thread placement.

The results are presented in Figure 3. *AsymSched* always performs close to the best static thread placement. In a few cases where it does not, the difference is not statistically significant. For applications that produce the highest degree of contention on the interconnect links (streamcluster, PCA, and FaceRec),

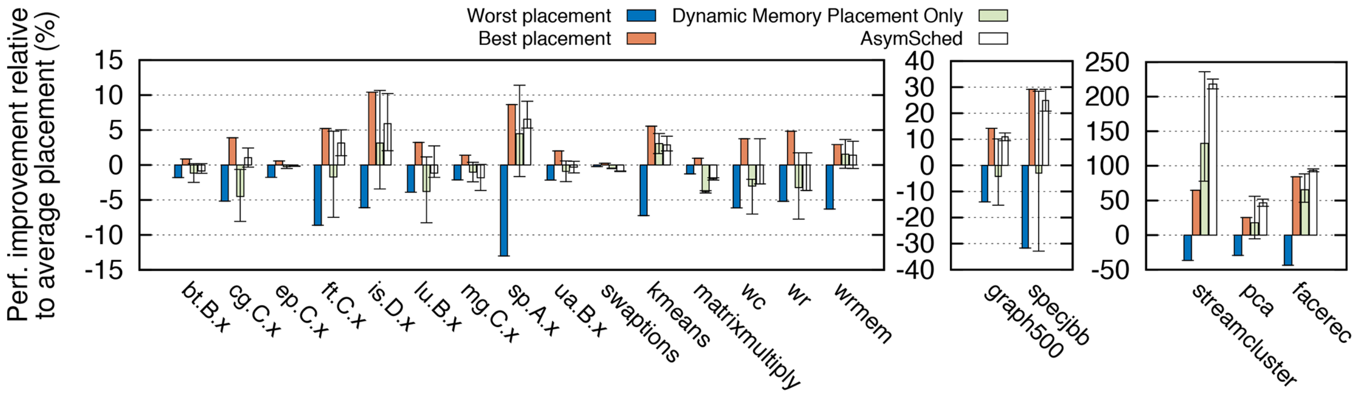


Figure 3: Performance difference between the best and worst static thread placement, dynamic memory placement, and *AsymSched* relative to the average thread placement on Machine A. Applications run with 24 threads on three nodes.

AsymSched achieves much better performance than the best thread placement, because the dynamic memory migration component balances memory accesses across nodes, thus reducing contention on interconnect links and memory controllers.

We also observe that dynamic memory migration without the migration of threads is not sufficient to achieve the best performance. More precisely, dynamic memory migration alone often achieves performance close to average. Moreover, it produces a high standard deviation for many benchmarks: the minimum and maximum performance often being the same as that of the best and worst static thread placement. For instance, on SPECjbb, the difference between the minimum and maximum performance with dynamic memory migration alone is 91%.

In contrast, *AsymSched* produces a very low standard deviation for most benchmarks. Two exceptions are is.D and SPECjbb. This is because in both cases, *AsymSched* migrates a large amount of memory. Both applications become memory intensive after an initialization phase, and *AsymSched* starts migrating memory only after the entire working set has been allocated. For instance, in the case of is.D, *AsymSched* migrates between 0 GB and 20 GB, depending on the initial placement of threads.

Conclusion

Asymmetry of the interconnect in modern NUMA systems drastically impacts performance. We found that the performance is more affected by the bandwidth between nodes than by the distance between them. We developed *AsymSched*, a new thread and memory placement algorithm that maximizes the bandwidth for communicating threads.

As the number of nodes in NUMA machines increases, the interconnect is less likely to remain symmetric. We believe that the clustering and placement techniques used in *AsymSched* will scale and be well adapted to these machines. Indeed, with very simple heuristics we were able to avoid computing up to 99% of the possible thread placements. Such optimizations will still likely be possible on future machines, as machines are usually made of multiple identical cores/sockets (e.g., our 64-core machine has four identical sockets). On machines that offer a wider diversity of thread placements, a possibility will be to use statistical approaches, such as that of Radojković et al. [12] to find good thread placements with a bounded overhead.

References

[1] Graph500 reference implementation: http://www.graph500.org/referencecode.

[2] SPECjbb2005 industry-standard server-side Java benchmark (j2se 5.0): http://www.spec.org/jbb2005/.

[3] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga, "The NAS Parallel Benchmarks Summary and Preliminary Results," in *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, pp. 158–165.

[4] S. Blagodurov, S. Zhuravlev, M. Dashti, and A. Fedorova, "A Case for NUMA-Aware Contention Management on Multicore Systems," in *Proceedings of the 2011 USENIX Annual Technical Conference (ATC '11)*, 2011.

[5] Timothy Brecht, "On the Importance of Parallel Application Placement in NUMA Multiprocessors," in *Proceedings of the USENIX Fourth Symposium on Experiences with Distributed and Multiprocessor Systems (SEDMS IV)*, 1993.

[6] CSU Face Identification Evaluation System: http://www.cs.colostate.edu/evalfacerec/index10.php.

[7] Mohammad Dashti, Alexandra Fedorova, Justin Funston, Fabien Gaud, Renaud Lachaize, Baptiste Lepers, Vivien Quéma, and Mark Roth, "Traffic Management: A Holistic Approach to Memory Placement on NUMA Systems," in *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ACM, 2013, pp. 381–394.

[8] Renaud Lachaize, Baptiste Lepers, and Vivien Quéma, "MemProf: A Memory Profiler for NUMA Multicore Systems," in *Proceedings of the 2012 USENIX Annual Technical Conference (ATC '12)*, 2012.

[9] Baptiste Lepers, Vivien Quéma, and Alexandra Fedorova, "Thread and Memory Placement on NUMA Systems: Asymmetry Matters," in *Proceedings of the 2015 USENIX Annual Technical Conference (ATC '15)*, Santa Clara, CA, July 2015, pp. 277–289.

[10] Metis MapReduce Library: http://pdos.csail.mit.edu/metis/.

[11] PARSEC Benchmark Suite: http://parsec.cs.princeton.edu/.

[12] Petar Radojković, Vladimir Cakarević, Miquel Moretó, Javier Verdú, Alex Pajuelo, Francisco J. Cazorla, Mario Nemirovsky, and Mateo Valero, "Optimal Task Assignment in Multithreaded Processors: A Statistical Approach," *SIGARCH Comput. Archit. News*, vol. 40, no. 1, March 2012, pp. 235–248.

[13] David Tam, Reza Azimi, and Michael Stumm, "Thread Clustering: Sharing-Aware Scheduling on SMP-CMP-SMT Multiprocessors," in *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys, 2007), pp. 47–58.*

[14] Lingjia Tang, J. Mars, Xiao Zhang, R. Hagmann, R. Hundt, and E. Tune, "Optimizing Google's Warehouse Scale Computers: The NUMA Experience," in *Proceedings of the IEEE 19th International Symposium on High Performance Computer Architecture (HPCA '13)*, Feb. 2013, pp. 188–197.

An Introduction to B$^\varepsilon$-trees and Write-Optimization

MICHAEL A. BENDER, MARTIN FARACH-COLTON, WILLIAM JANNEN, ROB JOHNSON, BRADLEY C. KUSZMAUL, DONALD E. PORTER, JUN YUAN, AND YANG ZHAN

Michael A. Bender is a Professor of Computer Science at Stony Brook University in Stony Brook, New York. His research focuses on algorithms, particularly on out-of-core algorithms. Bender co-founded the database company Tokutek, which was recently acquired by Percona. He has won several awards, including an R&D 100 award, a Test of Time award, a Best Paper award, a Best Newcomer award, and five teaching awards. bender@cs.stonybrook.edu

Martin Farach-Colton is a Professor of Computer Science at Rutgers University, New Brunswick, New Jersey. His research focuses on both the theory and practice of external memory and storage systems. He was a pioneer in the theory of cache oblivious analysis. His current research focuses on the use of write optimization to improve performance in both read- and write-intensive big data systems. He has also worked on the algorithmics of strings and metric spaces, with applications to bioinformatics. In addition to his academic work, Professor Farach-Colton has extensive industrial experience. He is CTO and co-founder of Tokutek, a database company that was founded to commercialize his research. During 2000–2002, he was a Senior Research Scientist at Google. farach@cs.rutgers.edu

William Jannen is a PhD student at Stony Brook University, where he attempts to design systems that accommodate the physical characteristics of their underlying media. He is also an artist and a player of games. wjannen@cs.stonybrook.edu

A B$^\varepsilon$-tree is an example of a *write-optimized* data structure and can be used to organize on-disk storage for an application, such as a database or file system. A B$^\varepsilon$-tree provides a key-value API, similar to a B-tree, but with better performance, particularly for inserts, range queries, and key-value updates. This article describes the B$^\varepsilon$-tree, compares its asymptotic performance to B-trees and Log-Structured Merge trees (LSM-trees), and presents real-world performance measurements. After finishing this article, a reader should have a basic understanding of how a B$^\varepsilon$-tree works, its performance characteristics, how it compares to other key-value stores, and how to design applications to gain the most performance from a B$^\varepsilon$-tree.

B$^\varepsilon$-trees

B$^\varepsilon$-trees were proposed by Brodal and Fagerberg [1] as a way to demonstrate an asymptotic performance tradeoff curve between B-trees [2] and buffered repository trees [3]. Both data structures support the same operations, but a B-tree favors queries, whereas a buffered repository tree favors inserts.

Researchers, including the authors of this article, have recognized the practical utility of a B$^\varepsilon$-tree when configured to occupy the "middle ground" of this curve—realizing query performance comparable to a B-tree but insert performance orders of magnitude faster than a B-tree. The B$^\varepsilon$-tree has since been used by both the high-performance, commercial TokuDB database [4] and the BetrFS research file system [5]. For the interested reader, we have created a simple, reference implementation of a B$^\varepsilon$-tree, available at https://github.com/oscarlab/Be-Tree.

We first explain how the basic operations on a B$^\varepsilon$-tree work. We then give the motivation behind these design choices and illustrate how these choices affect the asymptotic analysis.

API and basic structure. A B$^\varepsilon$-tree is a B-tree-like search tree for organizing on-disk data, as illustrated in Figure 1. Both B-trees and B$^\varepsilon$-trees export a key-value store API:

- insert(k, v)
- delete(k)
- v = query(k)
- $[v_1, v_2, ...]$ = range-query(k_1, k_2)

Like a B-tree, the node size in a B$^\varepsilon$-tree is chosen to be a multiple of the underlying storage device's block size. Typical B$^\varepsilon$-tree node sizes range from a few hundred kilobytes to a few megabytes. In both B-trees and B$^\varepsilon$-trees, internal nodes store pivot keys and child pointers, and leaves store key-value pairs, sorted by key. For simplicity, one can think of each key-value or pivot-pointer pair as being unit size; both B-trees and B$^\varepsilon$-trees can store keys and values of different sizes in practice. Thus, a leaf of size B holds B key-value pairs, which we call `items` below.

The distinguishing feature of a B$^\varepsilon$-tree is that internal nodes also allocate some space for a buffer, as shown in Figure 1. The buffer in each internal node is used to store `messages`, which encode updates that will eventually be applied to items in leaves under this node. This

Rob Johnson is a Research Professor at Stony Brook University and conducts research on security, big data algorithms, and cryptography. He does theoretical work with an impact on the real world. rob@cs.stonybrook.edu

Bradley C. Kuszmaul is a Research Scientist in the Computer Science and Artificial Intelligence Laboratory at the Massachusetts Institute of Technology (MIT CSAIL). His research focuses on performance engineering of multicore software as well as on data structures and algorithms that optimize cache and disk I/O. bradley@mit.edu

Donald E. Porter is an Assistant Professor of Computer Science at Stony Brook University in Stony Brook, New York. His research aims to improve computer system efficiency and security. In addition to recent work on write optimization in file systems, recent projects have developed lightweight guest operating systems for virtual environments, system security abstractions, and efficient data structures for caching. porter@cs.stonybrook.edu

Jun Yuan is a PhD student in computer science at Stony Brook University in Stony Brook, New York. Her research interest primarily lies in compiler and system security. In addition to write-optimized file systems, she has recently studied access control on the Android OS. junyuan@cs.stonybrook.edu

Yang Zhan is a PhD student in the Department of Computer Science at Stony Brook University. His research interests include file system and system performance. yazhan@cs.stonybrook.edu

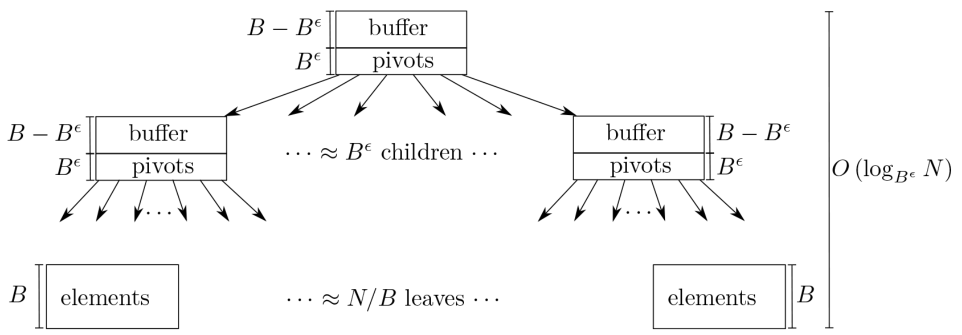

$B - B^\epsilon$
buffer
B^ϵ
pivots
$\cdots \approx B^\epsilon$ children $\cdots$
$O(\log_{B^\epsilon} N)$
B
elements
$\cdots \approx N/B$ leaves $\cdots$

Figure 1: A B$^\varepsilon$-tree. Each node is roughly of size B, and ε controls how much of an internal node's space is used for pivots (B^ε) and how much is used for buffering pending updates ($B - B^\varepsilon$). As in a B-tree, items are stored in leaves, and the height of the tree is logarithmic in the total number of items (N), based on the branching factor (here B^ε).

buffer is not an in-memory data structure; it is part of the node and is written to disk, evicted from memory, etc., whenever the node is. The value of ε, which must be between 0 and 1, is a tuning parameter that selects how much space internal nodes use for pivots ($\approx B^\varepsilon$) and how much space is used as a buffer ($\approx B - B^\varepsilon$).

Inserts and deletes. Insertions are encoded as "insert messages," addressed to a particular key and added to the buffer of the root node of the tree. When enough messages have been added to a node to fill the node's buffer, a batch of messages are flushed to one of the node's children. Generally, the child with the most pending messages is selected. Over the course of flushing, each message is ultimately delivered to the appropriate leaf node, and the new key and value are added to the leaf. When a leaf node becomes too full, it splits, just as in a B-tree. Similar to a B-tree, when an interior node gets too many children, it splits and the messages in its buffer are distributed between the two new nodes.

Moving messages down the tree in batches is the key to the B$^\varepsilon$-tree's insert performance. By storing newly inserted messages in a buffer near the root, a B$^\varepsilon$-tree can avoid seeking all over the disk to put elements in their target locations. The B$^\varepsilon$-tree only moves messages to a subtree when enough messages have accumulated for that subtree to amortize the I/O cost. Although this involves rewriting the same data multiple times, this can *improve* performance for smaller, random inserts, as our analysis in the next section shows.

B$^\varepsilon$-trees delete items by inserting "tombstone messages" into the tree. These tombstone messages are flushed down the tree until they reach a leaf. When a tombstone message is flushed to a leaf, the B$^\varepsilon$-tree discards both the deleted item and the tombstone message. Thus, a deleted item, or even entire leaf node, can continue to exist until a tombstone message reaches the leaf. Because deletes are encoded as messages, deletions are algorithmically very similar to insertions.

A high-performance B$^\varepsilon$-tree should detect and optimize the case where a large series of messages all go to one leaf. Suppose a series of keys are inserted that will completely fill one leaf. Rather than write these messages to an internal node only to immediately rewrite them to each node on the path from root to leaf, these messages should flush directly to the leaf, along with any other pending messages for that leaf. The B$^\varepsilon$-tree implementation in TokuDB and BetrFS includes some heuristics to avoid writing to intermediate nodes when a batch of messages are all going to a single child.

Point and range queries. Messages addressed to a key k are guaranteed to be applied to k's leaf or in some buffer along the root-to-leaf path towards key k. This invariant ensures that point and range queries in a B$^\varepsilon$-tree have a similar I/O cost to a B-tree.

In both a B-tree and a B$^\varepsilon$-tree, a point query visits each node from the root to the correct leaf. However, in a B$^\varepsilon$-tree, answering a query also means checking the buffers in nodes on this path for messages, and applying relevant messages before returning the results of the query. For example, if a query for key k finds an entry (k,v) in a leaf and a tombstone message for k in the buffer of an internal node, then the query will return "NOT FOUND", since the entry for key k has been logically deleted from the tree. Note that the query need not update the leaf in this case—it will eventually be updated when the tombstone message is flushed to the leaf. A range query is similar to a point query, except that messages for the entire range of keys must be checked and applied as the appropriate subtree is traversed.

In order to make searching and inserting into buffers efficient, the message buffers within each node are typically organized into a balanced binary search tree, such as a red-black tree. Messages in the buffer are sorted by their target key, followed by timestamp. The timestamp ensures that messages are applied in the correct order. Thus, inserting a message into a buffer, searching within a buffer, and flushing from one buffer to another are all fast.

Performance Analysis

We analyze the behavior of B-trees, B$^\varepsilon$-trees, and LSM-trees in this article in terms of I/Os. Our primary interest is in data sets too large to fit into RAM. Thus, the first-order performance impact is how many I/O requests must be issued to complete each operation. In the algorithms literature, this is known as the disk-access-machine (DAM) model, the external-memory model, or the I/O model [6].

Performance model. In order to compare B-trees and B$^\varepsilon$-trees in a single framework, we make a few simplifying assumptions. We assume that all key-value pairs are the same size and that each node in the tree can hold B key-value pairs. The entire tree stores N key-value pairs. We also assume that each node can be accessed with a single I/O transaction—i.e., on a rotating disk, the node is stored contiguously and requires only one random seek.

This model focuses on the principal performance characteristics of a block storage device, such as a hard drive or SSD. For instance, on a hard drive, this model captures the latency of a random seek to read a node. In the case of an SSD, the model captures the I/O bandwidth costs, i.e., the number of blocks that must be read or written from the device per operation. Regardless of whether the device is bandwidth or latency bound, for a given node size B, minimizing the number of nodes accessed minimizes both bandwidth and latency costs.

B$^\varepsilon$-tree I/O performance. Table 1 lists the asymptotic complexities of each operation in a B-tree and B$^\varepsilon$-tree. We will explain upserts and epsilon (ε), as well as how they affect performance, later in the article. For this discussion, note that ε is a tuning parameter between 0 and 1; ε is generally set at design time and becomes a constant in the analysis.

The point-query complexities of a B-tree and a B$^\varepsilon$-tree are both logarithmic in the number of items ($O(\log_B N)$); a B$^\varepsilon$-tree adds a constant overhead of $1/\varepsilon$. Compared to a B-tree with the same node size, a B$^\varepsilon$-tree reduces the fanout from B to B$^\varepsilon$, making the tree taller by a factor of $1/\varepsilon$. Thus, for example, querying a B$^\varepsilon$-tree where $\varepsilon = 1/2$ will require, at most, twice as many I/Os.

Range queries incur a logarithmic search cost for the first key, as well as a cost that is proportional to the size of the range and how many disk blocks the range is distributed across. The scan cost is roughly the number of keys read (k) divided by the block size (B). The total cost of a range query is $O(k/B + \log_B N)$ I/Os.

Compared to a B-tree, batching messages *divides the insertion cost by the batch size* ($B^{1-\varepsilon}$). For example, if $B = 1024$ and $\varepsilon = 1/2$, a B$^\varepsilon$-tree can perform inserts $\approx \varepsilon B^{1-\varepsilon} = \frac{1}{2}\sqrt{1024} = 16$ times faster than a B-tree.

Write optimization. Batching small, random inserts is an essential feature of write-optimized data structures (WODS), such as a B$^\varepsilon$-tree or LSM-tree. Although a WODS may issue a small write multiple times as a message moves down the tree, once the I/O cost is divided among a large batch, the cost per insert or delete is much smaller than one I/O per operation. In contrast, a workload of random inserts to a B-tree requires a *minimum* of one I/O per insert—to write the new element to its target leaf.

The B$^\varepsilon$-tree flushing strategy is designed to ensure that it can always move elements in large batches. Messages are only flushed to a child when the buffer of a node is full, containing $B - B^\varepsilon \approx B$ messages. When a buffer is flushed, not all messages are necessarily flushed—messages are only flushed to children with enough pending messages to offset the cost of rewriting the parent and child nodes. Specifically, at least $(B - B^\varepsilon)/B^\varepsilon \approx B^{1-\varepsilon}$ messages are moved from the parent's buffer to the child's on each flush. Consequently, any node in a B$^\varepsilon$-tree is only rewritten if a sufficiently large portion of the node will change.

Caching. Most systems cache a subset of the tree in RAM. With an LRU replacement policy, accesses to the top of the tree are likely to hit in the cache, whereas accesses to leaves and "lower nodes" will more commonly miss. Thus, when the cache is warm, the actual cost of a search may be much less than $O(\log_B N)$ I/Os. For both B-trees and B$^\varepsilon$-trees, if only the leaves are out-of-cache, point queries and updates require a single I/O, whereas a range query has an I/O cost that is linear in the number of leaves read.

The Impact of Node Size (*B*) on Performance

B-trees have small nodes to balance the cost of insertions and range queries. B-tree implementations face a tradeoff between update and range-query performance. A larger node size B favors range queries, and a smaller node size favors inserts and deletes. Larger nodes help range-query performance because the I/O costs, such as seeks, can be amortized over more data. However, larger nodes make updates more expensive because a leaf node and possibly internal nodes must be completely rewritten each time a new item is added to the index, and larger nodes mean more to rewrite.

Thus, many B-tree implementations use small nodes (tens to hundreds of KB), resulting in sub-optimal range-query performance. As free space on disk becomes fragmented, B-tree nodes may also become scattered on disk; this is sometimes called *aging*. Now a range query must seek for each leaf in the scan, resulting in poor bandwidth utilization.

For example, with 4 KB nodes stored on a disk with a 5 ms seek time and 100 MB/s bandwidth, updating a single key only rewrites 4 KB. Range queries, however, must perform a seek for each 4 KB leaf node, resulting in a net bandwidth of 800 KB/s, less than 1% of the disk's potential bandwidth.

B$^\varepsilon$-trees have efficient updates and range queries even when nodes are large. In contrast, batching in a B$^\varepsilon$-tree allows B to be much larger in a B$^\varepsilon$-tree than in a B-tree. In a B$^\varepsilon$-tree the bandwidth cost per insert is $O(\frac{\log_B N}{\varepsilon B^{1-\varepsilon}})$, which grows much more slowly as B increases. As a result, B$^\varepsilon$-trees use node sizes of a few hundred kilobytes to a few megabytes.

By using large B, B$^\varepsilon$-trees can perform range queries at near disk bandwidth. For example, a B$^\varepsilon$-tree using 4 MB nodes need perform only one seek for every 4 MB of data it returns, yielding a net bandwidth of over 88 MB/s on the same disk as above.

In the comparison of insert complexities above, we stated that a B$^\varepsilon$-tree with $\varepsilon = 1/2$ would be twice as deep as a B-tree, as some fanout is sacrificed for buffer space. This is only true when the node size is the same. Because a B$^\varepsilon$-tree can use larger nodes in practice, a B$^\varepsilon$-tree can still have close to the same fanout and height as a B-tree.

The Role of ε

The parameter ε in a B$^\varepsilon$-tree was originally designed to show that there is an optimal tradeoff curve between insert and point query performance. Parameter ε ranges between 0 and 1. As we explain in the rest of this section, making ε an exponent simplifies the asymptotic analysis and creates an interesting tradeoff curve.

Intuitively, the tradeoff with parameter ε is how much space in the node is used for storing pivots and child pointers ($\approx B^\varepsilon$) and how much space is used for message buffers ($\approx B - B^\varepsilon$). As ε increases, so does the branching factor (B^ε), causing the depth of the tree to decrease and searches to run faster. As ε decreases, the buffers get larger, batching more inserts for every flush and improving overall insert performance.

At one extreme, when $\varepsilon = 1$, a B$^\varepsilon$-tree is just a B-tree, since interior nodes contain only pivot keys and child pointers. At the other extreme, when $\varepsilon = 0$, a B$^\varepsilon$-tree is a binary search tree with a large buffer at each node, called a buffered repository tree [3].

The most interesting configurations place ε strictly between 0 and 1, such as $\varepsilon = 1/2$. For such configurations, a B$^\varepsilon$-tree has the same asymptotic point query performance as a B-tree, but asymptotically better insert performance.

For inserts, setting $\varepsilon = 1/2$ *divides* the cost by the square root of node size. Formally, the cost then becomes: $O(\frac{\log_B N}{\varepsilon B^{1-\varepsilon}}) = O(\frac{\log_B N}{\sqrt{B}})$. Since the insert cost is divided by $\sqrt{B}$, selecting larger nodes (increasing B) can dramatically improve insert performance.

Assuming all other parameters are the same, decreasing ε slows down point queries by a constant $1/\varepsilon$. To see the query performance for $\varepsilon = 1/2$, evaluate the point query cost in Table 1: $O(\frac{\log_B N}{\varepsilon}) = O(\frac{\log_B N}{1/2}) = O(2 \log_B N)$—doubling the number of I/Os. Changing ε from 1/2 to 1/4 would make this a factor of 4. This cost can be offset by increasing B, which, as pointed out above, also improves insert performance.

The above analysis assumes all keys have unit size and that nodes can hold B keys; real systems must deal with variable-sized keys, so B, and hence ε, are not fixed or known a priori. Nonetheless, the main insight of B$^\varepsilon$-trees—that we can speed up insertions by buffering items in internal nodes and flushing them down the tree in batches—still applies in this setting.

Data Structure	Insert	Point Query no Upserts	Point Query w/ Upserts	Range Query
B$^\varepsilon$-tree	$\frac{\log_B N}{\epsilon B^{1-\epsilon}}$	$\frac{\log_B N}{\epsilon}$	$\frac{\log_B N}{\epsilon}$	$\frac{\log_B N}{\epsilon} + \frac{k}{B}$
B$^\varepsilon$-tree ($\epsilon = 1/2$)	$\frac{\log_B N}{\sqrt{B}}$	$\log_B N$	$\log_B N$	$\log_B N + \frac{k}{B}$
B-tree	$\log_B N$	$\log_B N$	$\log_B N$	$\log_B N + \frac{k}{B}$
LSM	$\frac{\log_B N}{\epsilon B^{1-\epsilon}}$	$\frac{\log_B^2 N}{\epsilon}$	$\frac{\log_B^2 N}{\epsilon}$	$\frac{\log_B^2 N}{\epsilon} + \frac{k}{B}$
LSM+BF	$\frac{\log_B N}{\epsilon B^{1-\epsilon}}$	$\log_B N$	$\frac{\log_B^2 N}{\epsilon}$	$\frac{\log_B^2 N}{\epsilon} + \frac{k}{B}$

Table 1: Asymptotic I/O costs of important operations. B$^\varepsilon$-trees simultaneously support efficient inserts, point queries (even in the presence of upserts), and range queries. These complexities apply for $0 < \varepsilon \leq 1$. Note that ε is a design-time constant. We show the complexity for general ε and evaluate the complexity when ε is set to a typical value of 1/2. The $1/\varepsilon$ factor evaluates to a constant that disappears in the asymptotic analysis.

In practice, B^ε-tree implementations select a fixed physical node size and fanout (B^ε). For the implementation in TokuDB and BetrFS, nodes are approximately 4 MB, and the branching factor ranges from 4 to 16. As a result, the B^ε-tree can always flush data in batches of at least 256 KB.

How to Speed up Applications by Using a B^ε-tree

A practical consequence of the analysis above is that a B^ε-tree can perform updates orders of magnitude faster than point queries. This search-insert asymmetry has two implications for designing applications on B^ε-trees.

Performance rule. *Avoid query-before-update whenever possible.*

Because of the search-insert asymmetry, the common read-modify-write (or query-modify-insert) pattern will be bound to the speed of a query, which is no faster in a B^ε-tree than in a B-tree.

Upserts. B^ε-trees support a new upsert operation, to help applications bridge this performance asymmetry. An upsert is a type of message that encodes an update with a callback function which does not require first knowing the value of the key.

Upserts can encode any modification that is asynchronous and depends only on the key, the old value, and some auxiliary data that can be stored with the upsert message. Tombstones are a special case of upserts. Upserts can also be used to increment a counter, update the access time on a file, update a user's account balance after a withdrawal, and many other operations.

With upserts, an application can update the value associated with key k in the B^ε-tree by inserting an "upsert message" $(k, (f, \Delta))$ into the tree, where f is a call-back function and Δ is auxiliary data specifying the update to be performed. This upsert message is semantically equivalent to performing a query followed by an insert:

$$v \leftarrow \text{query}(k); \text{insert}(k, f(v, \Delta)).$$

However, the upsert does not perform these operations. Rather, the message $(k, (f, \Delta))$ is inserted into the tree like an insert or tombstone message.

When an upsert message $(k, (f, \Delta))$ is flushed to a leaf, the value v associated with k in the leaf is replaced by $f(v, \Delta)$ and the upsert message is discarded. If the application queries k before the upsert message reaches a leaf, then the upsert message is applied to v before the query returns.

As with any insert or delete message, multiple upserts can be buffered for the same key between the root and leaf. If a key is queried with multiple upserts pending, each upsert must be collected on the path from root to leaf and applied to the key in the order they were inserted into the tree.

The upsert mechanism does not interfere with I/O performance of searches, because the upsert messages for a key k always lie on the search path from the root of the B^ε-tree to the leaf containing k. Thus, the upsert mechanism can accelerate updates by one to two orders of magnitude without slowing down queries.

Performance rule. *Use insert performance to improve query performance by maintaining appropriate indices.*

Secondary indices. In a database, secondary indices can greatly speed up queries. For example, consider a database table with three columns, k_1, k_2, and k_3, and an application that sometimes performs queries using k_1 and sometimes using k_2. If the table is implemented as a B-tree sorted on k_1, then queries using k_1 are fast, but queries using k_2 are extremely slow—they may have to scan essentially the entire database. To solve this problem, the table can be configured to maintain two indices—one sorted by k_1 and one sorted by k_2. Queries can then use the appropriate index based on the type of the query.

When multiple indices are maintained with B-trees, each index update requires an additional insert. Because inserts are as expensive as a point query, maintaining an index on each column is often impractical. Thus, the table designer must carefully analyze factors such as the expected type of queries and distribution of keys in deciding which columns to index, in order to ensure good overall performance.

B^ε-trees turn these issues upside down. Indices are cheap to maintain. Point queries are fundamentally expensive—B^ε-tree point queries are no faster than in a B-tree. Thus, B^ε-tree applications should maintain whatever indices are needed to perform queries efficiently.

There are three rules for designing good B^ε-tree indices.

First, maintain indices sorted by the keys used to query the database. For example, in the above example, the database should maintain two B^ε-trees—one sorted by k_1 and one sorted by k_2.

Second, ensure that each index has all the information required to answer the intended queries. For example, if the application looks up the k_3 value using key k_2, then the index sorted by k_2 should store the corresponding k_3 value for each entry. In many databases, the secondary index contains only keys into the primary index. Thus, for example, a query on k_2 would return the primary key value, k_1. To obtain k_3, the application has to perform another query in the primary index using the k_1 value obtained from the secondary index. An index that contains all the information relevant to a query is called a covering index for that query.

Finally, design indices to enable applications to perform range queries whenever possible. For example, if the application wants to look up all entries (k_1, k_2, k_3) for which $a \leq k_1 \leq b$, and k_2 satisfies some predicate, then the application should maintain a secondary index sorted by k_1 that only contains entries for which k_2 matches the predicate.

Log-Structured Merge-Trees

Log-structured merge trees (LSM-trees) [7] are a WODS with many variants [8, 9]. An LSM-tree typically consists of a logarithmic number of B-trees of exponentially increasing size. Once an index at one level fills up, it is emptied by merging it into the index at the next level. The factor by which each level grows is a tunable parameter comparable to the branching factor (B$^\varepsilon$) in a B$^\varepsilon$-tree. For ease of comparison, Table 1 gives the I/O complexities of operations in an LSM-tree with growth factor B$^\varepsilon$.

LSM-trees can be tuned to have the same insertion complexity as a B$^\varepsilon$-tree, but queries in a naïvely implemented LSM-tree can require $O(\frac{\log_B^2 N}{\varepsilon})$ I/Os because the query must be repeated in $O(\log_B N)$ B-trees. Most LSM-tree implementations use Bloom filters to avoid queries in all but one of the B-trees, improving point query performance to $O(\frac{\log_B N}{\varepsilon})$ I/Os.

One problem for LSM-trees is that the benefits of Bloom filters do not extend to range queries. Bloom filters are only designed to improve point queries and do not support range queries. Thus, a range query must be done on every level of the LSM-tree—squaring the search overhead in Table 1 and yielding strictly worse asymptotic performance than a B$^\varepsilon$-tree or a B-tree.

A second advantage of a B$^\varepsilon$-tree over an LSM-tree is that B$^\varepsilon$-trees can effectively use upserts, whereas upserts in an LSM-tree will ruin the performance advantage of adding Bloom filters. As discussed above, upserts address a search-insert asymmetry common to any WODS, including LSM-trees. When an application uses upserts, it is possible for a message for that key to be present in every level of the tree containing a pending message for the key. Thus, a subsequent point query will still have to query every level of the tree, defeating the purpose of adding Bloom filters. Note that querying every level of an LSM-tree also squares the overhead compared to a B$^\varepsilon$-tree or B-tree, and is more expensive than walking the path from root-to-leaf in a B$^\varepsilon$-tree.

In summary, Bloom-filter-enhanced LSM-trees can match the performance of B$^\varepsilon$-trees for some but not all workloads. B$^\varepsilon$-trees asymptotically dominate LSM-tree performance. In particular, B$^\varepsilon$-trees are asymptotically faster than LSM-trees for small range queries and point queries in upsert-intensive workloads.

Performance Comparison

To give a sense of how B$^\varepsilon$-trees perform in practice, we present some data from BetrFS, an in-kernel, research file system based on B$^\varepsilon$-trees. We compare BetrFS to other file systems, including

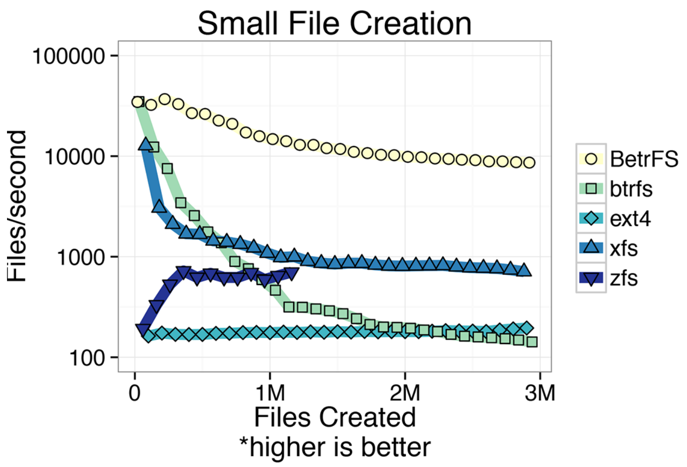


Figure 2: Sustained rate of file creation for 3 million 200-byte files, using four threads. Higher is better.

Btrfs, which is built with B-trees. A more thorough evaluation appears in our recent FAST paper [5].

All experimental results were collected on a Dell Optiplex 790 with a four-core 3.40 GHz Intel Core i7 CPU, 4 GB RAM, and a 250 GB, 7200 RPM ATA disk. Each file system used a 4096-byte block size. The system ran Ubuntu 13.10, 64-bit, with Linux kernel version 3.11.10. Each experiment compared several general-purpose file systems, including Btrfs, ext4, XFS, and ZFS. Error bars and ± ranges denote 95% confidence intervals. Unless otherwise noted, benchmarks are cold-cache tests.

Small writes. We used the TokuBench benchmark [10] to create 3 million 200-byte files in a balanced directory tree with fanout of 128, using four threads (one per CPU). In BetrFS, file creations are implemented as B$^\varepsilon$-tree inserts, and small file writes are implemented using upserts, so this benchmark demonstrates the B$^\varepsilon$-tree's performance on these two operations. Figure 2 shows the sustained rate of file creation in each file system (note the log scale). In the case of ZFS, the file system crashed before completing the benchmark, so we reran the experiment five times and used data from the longest-running iteration. BetrFS is initially among the fastest file systems, and continues to perform well for the duration of the experiment. The steady-state performance of BetrFS is an order of magnitude faster than the other file systems.

This performance distinction is attributable to both fewer total writes and fewer seeks per byte written—i.e., better aggregation of small writes. Based on profiling from blktrace, one major distinction is total bytes written: BetrFS writes 4–10x fewer total MB to disk, with an order of magnitude fewer total write requests. Among the other file systems, ext4, XFS, and ZFS wrote roughly the same amount of data, but realized widely varying underlying write throughput.

FS	find	grep
BetrFS	0.36 ± 0.06	3.95 ± 0.28
Btrfs	3.87 ± 0.94	14.91 ± 1.18
ext4	2.47 ± 0.07	46.73 ± 3.86
XFS	19.07 ± 3.38	66.20 ± 15.99
ZFS	11.60 ± 0.81	41.74 ± 0.64

Table 2: Directory operation benchmarks, measured in seconds. Lower is better.

Locality and directory operations. In BetrFS, fast range queries translate to fast large directory scans. Table 2 reports the time taken to run "find" and "grep -r" on the Linux 3.11.10 source tree, starting from a cold cache. The grep test recursively searches the file contents for the string "cpu_to_be64", and the find test searches for files named "wait.c".

Both the find and grep benchmarks do well because file system data and metadata are stored in large nodes and sorted lexicographically by full path. Thus, related files are stored near each other on disk. BetrFS also maintains a second index that contains only metadata, so that metadata scans can be implemented as range queries. As a result, BetrFS can search directory metadata and file data one or two orders of magnitude faster than the other file systems.

Limitations. It is important to note that BetrFS is a still a research prototype and currently has three primary cases where it performs considerably worse than other file systems: large directory renames, large deletes, and large sequential writes (more details in [5]). Renames and deletes are slow because BetrFS does not map them directly onto B^ε-tree operations. Sequential writes are slow largely because the underlying B^ε-tree appends all data to a log before inserting it into the tree, so everything is written to disk at least twice. We believe these issues can be addressed in ongoing research and development efforts; our goal, supported by the asymptotic analysis, is for BetrFS to match or exceed the performance of other file systems on all workloads.

Conclusion

B^ε-tree implementations can match the search performance of B-trees, perform inserts and delete orders of magnitude faster, and execute range queries at near disk bandwidth. The design and implementation of B^ε-trees converts a tradeoff between update and range query costs into a mutually beneficial synergy between batching small updates and large nodes. Our results with BetrFS demonstrate that the asymptotic improvements of B^ε-trees can yield practical performance improvements for applications that are designed for B^ε-tree's performance characteristics.

Acknowledgments

This work was supported in part by NSF grants CNS-1409238, CNS-1408782, CNS-1408695, CNS-1405641, CNS-1149229, CNS-1161541, CNS-1228839, CNS-1408782, IIS-1247750, CCF-1314547, Sandia National Laboratories, and the Office of the Vice President for Research at Stony Brook University.

References

[1] G. S. Brodal and R. Fagerberg, "Lower Bounds for External Memory Dictionaries," in *Proceedings of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms (ACM)*, 2003, pp. 546–554.

[2] D. Comer, "The Ubiquitous B-tree," *ACM Computing Surveys*, vol. 11, June 1979, pp. 121–137.

[3] A. L. Buchsbaum, M. Goldwasser, S. Venkatasubramanian, and J. R. Westbrook, "On External Memory Graph Traversal," in *Proceedings of the 11th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2000, pp. 859–860.

[4] Tokutek, Inc., TokuDB: MySQL Performance, MariaDB Performance, 2013: http://www.tokutek.com/products/tokudb-for-mysql/.

[5] W. Jannen, J. Yuan, Y. Zhan, A. Akshintala, J. Esmet, Y. Jiao, A. Mittal, P. Pandey, P. Reddy, L. Walsh, M. Bender, M. Farach-Colton, R. Johnson, B. C. Kuszmaul, and D. E. Porter, "BetrFS: A Right-Optimized Write-Optimized File System," in *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, 2015, pp. 301–315.

[6] A. Aggarwal and J. S. Vitter, "The Input/Output Complexity of Sorting and Related Problems," *Communications of the ACM*, vol. 31, Sept. 1988, pp. 1116–1127.

[7] P. O'Neil, E. Cheng, D. Gawlic, and E. O'Neil, "The Log-Structured Merge-Tree (LSM-tree)," *Acta Informatica*, vol. 33, no. 4, 1996, pp. 351–385.

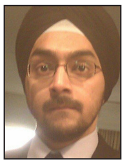
[8] R. Sears and R. Ramakrishnan, "bLSM: A General Purpose Log Structured Merge Tree," in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, ACM, 2012, pp. 217–228.

[9] P. Shetty, R. P. Spillane, R. Malpani, B. Andrews, J. Seyster, and E. Zadok, "Building Workload-Independent Storage with VT-trees," in *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, 2013, pp. 17–30.

[10] J. Esmet, M. A. Bender, M. Farach-Colton, and B. C. Kuszmaul, "The TokuFS Streaming File System," in *Proceedings of the 4th USENIX Workshop on Hot Topics in Storage (HotStorage)*, June 2012.

It's Time to End Monolithic Apps for Connected Devices

RAYMAN PREET SINGH, CHENGUANG SHEN, AMAR PHANISHAYEE, AMAN KANSAL, AND RATUL MAHAJAN



Rayman Preet Singh is a PhD candidate in computer science at the University of Waterloo, Canada. He is co-advised by S. Keshav and Tim Brecht, and has broad research interests in distributed systems and ubiquitous computing. rmmathar@uwaterloo.ca

Chenguang Shen is a PhD candidate in computer science at the University of California, Los Angeles (UCLA), working with Professor Mani Srivastava. He obtained his MS in computer science from UCLA in 2014, and a BEng in software engineering from Fudan University, Shanghai, China in 2012. Chenguang's research focuses on developing a mobile sensing framework for context awareness. cgshen@cs.ucla.edu

Amar Phanishayee is a Researcher at Microsoft Research. His research efforts center around rethinking the design of datacenter-based systems, from infrastructure for compute, storage, and networking to distributed systems that are scalable, robust to failures, and resource-efficient. He is also interested in storage and programming abstractions for connected devices. Amar received his PhD from Carnegie Mellon University in 2012. amar@microsoft.com

The proliferation of connected sensing devices (or Internet of Things) can in theory enable a range of "smart" applications that make rich inferences about users and their environment. But in practice, developing such applications today is arduous because they are constructed as monolithic silos, tightly coupled to sensing devices, and must implement all data sensing and inference logic, even as devices move or are temporarily disconnected. We present Beam, a framework and runtime for distributed inference-driven applications that breaks down application silos by decoupling their inference logic from other functionality. It simplifies applications by letting them specify "what should be sensed or inferred," without worrying about "how it is sensed or inferred." We discuss the challenges and opportunities in building such an inference framework.

Connected sensing devices such as cameras, thermostats, and in-home motion, door-window, energy, and water sensors, collectively dubbed the Internet of Things (IoT), are rapidly permeating our living environments, with an estimated 50 billion such devices projected for use by 2020 [2]. They enable a wide variety of applications spanning security, efficiency, healthcare, and others. Typically, these applications collect data using sensing devices to draw inferences about the environment or the user, and use these inferences to perform certain actions. For example, Nest uses motion sensor data to infer and predict home occupancy and adjusts the thermostat accordingly.

Most IoT applications today are being built in a monolithic way. That is, applications are tightly coupled to the hardware. For instance, Nest's occupancy prediction can only be used with the Nest device. Applications need to implement all the data collection, inferencing, and user functionality-related logic. For application developers, this increases the complexity of development, and hinders broad distribution of their applications because the cost of deploying their specific hardware limits user adoption. For end users, each sensing device they install is limited to a small set of applications, even though the hardware capabilities may be useful for a broader set of applications. How do we break free from this monolithic and restrictive setting? Can we enable applications to be programmed to work seamlessly in heterogeneous environments with different types of connected sensors and devices, while leveraging devices that may only be available opportunistically, such as smartphones and tablets?

To address this problem, we start from the insight that many inferences required by applications can be drawn using multiple types of connected devices. For instance, home occupancy can be inferred using motion sensors (e.g., those in security systems or in Nest), cameras (e.g., Dropcam), microphone, smartphone GPS, or using a combination of these, since each may have different sources of errors. Therefore, *we posit that inference logic, traditionally left up to applications, ought to be abstracted out as a system service*. Such a service will relieve application developers of the burden of implementing and training commonly used

Aman Kansal received his PhD in electrical engineering from the University of California Los Angeles, where he was honored with the department's Outstanding PhD Award. His current research interests include all aspects of sensing systems, with a special emphasis on embedded sensing, context inference, and energy efficiency. He has published over 65 papers, and his work has also been recognized with the Microsoft Gold Star award. kansal@microsoft.com

Ratul Mahajan is a Principal Researcher at Microsoft Research and an Affiliate Professor at the University of Washington. His research interests include all aspects of networked systems. His current work focuses on software-defined networks and network verification, and his past work spans Internet routing and measurements, incentive-compatible protocol design, practical models for wireless networks, vehicular networks, and connected homes. ratul@microsoft.com

inferences. More importantly, it will enable applications to work using any of the sensing devices that the shared inference logic supports.

We surveyed and analyzed two popular application *classes* in detail, one that infers environmental attributes and another that senses an individual user.

- *Rules*: A large class of applications is based on the *If This Then That* (IFTTT) pattern [1, 8]. IFTTT enables users to create their own rules that map sensed attributes to desired actions. We consider a particular rules application that alerts a user if a high-power appliance, e.g., electric oven, is left on when the home is unoccupied. This application uses the appliance-state and home occupancy inferences.
- *Quantified Self* (QS) captures a popular class of applications that disaggregate a user's daily routine by tracking her physical activity (walking, running, etc.), social interactions (loneliness), mood (bored, focused), computer use, and more.

In analyzing these two popular classes of applications, we identify the following three key challenges for the proposed inference service:

1. Decouple applications, inference algorithms, and devices: Data-driven inferences can often be derived using data from multiple devices. Combining inputs from multiple devices, when available, generally leads to improved inference accuracy (often overlooked by developers). Figure 1 illustrates the improvement in inference accuracy for the occupancy and physical activity inferences, used in the Rules and Quantified Self applications, respectively; 100% accuracy maps to manually logged ground truth over 28 hours.

Hence, applications should not be restricted to using a single sensor or a single inference algorithm. At the same time, applications should not be required to incorporate device discovery, handle the challenges of potentially using devices over the wide area (e.g., remote I/O and tolerating disconnections), use disparate device APIs, and instantiate and combine multiple inferences depending on available devices. Therefore, an inference framework must decouple (1) "what is sensed" from "how it is sensed" and (2) "what is inferred" from "how it is inferred." It should require an application to only specify the desired inference, e.g., occupancy (in addition to inference parameters like sampling rate and coverage), while handling the complexity of configuring the right devices and inference algorithms.

2. Handle environmental dynamics: Applications are often interested in tracking user and device mobility, and adapting their processing accordingly. For instance, the QS application needs to track which locations a user frequents (e.g., home, office, car, gym, meeting room, etc.), handle intermittent connectivity, and more. Application development stands to be greatly simplified if the framework were to seamlessly handle such environmental dynamics, e.g., automatically update the selection of devices used for drawing inferences based on user location. Hence the QS application, potentially running on a cloud server, could simply subscribe to the activity inference, which would be dynamically composed of sub-inferences from various devices tracking a user.

3. Optimize resource usage: Applications often involve continuous sensing and inferring, and hence consume varying amounts of system resources across multiple devices over time. Such an application must structure its sensing and inference processing across multiple devices, in keeping with the devices' resource constraints. This adds undue burden on developers. For instance, in the QS application, wide area bandwidth constraints may prevent backhauling of high rate sensor data. Moreover, whenever possible, inferences should be shared across multiple applications to prevent redundant resource consumption. Therefore, an inference framework must not only facilitate sharing of inferences, but in doing so must optimize for efficient resource use (e.g., network, battery, CPU, memory, etc.) while meeting resource constraints.

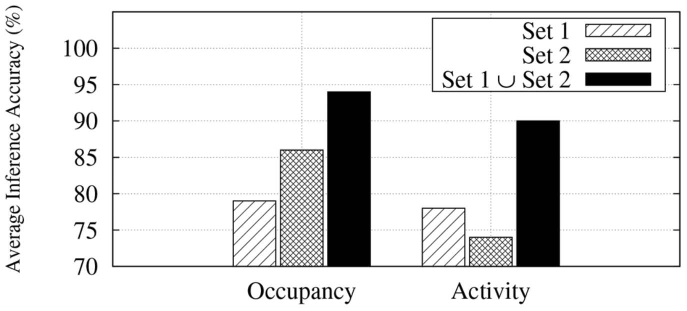


Figure 1: Improvement in occupancy and activity inference accuracy by combining multiple devices. For occupancy, sensor set 1 = {camera, microphone} in one room and set 2 = {PC interactivity detection} in a second room. For physical activity, set 1 = {phone accelerometer} and set 2 = {wrist worn FitBit}.

Beam Inference Framework

To explore the above challenges concretely, we propose Beam, an application framework and associated runtime for data-driven inference-based applications. Beam provides applications with inference-based programming abstractions. Applications subscribe to high-level inferences, and Beam dynamically identifies the required sensors in the given deployment and constructs an appropriate *inference graph*. The inference graph is made up of *modules,* which are processing units that encapsulate inference algorithms; modules can use the output of other modules for their processing logic. The Beam runtime instantiates the inference graph to initiate data processing on suitable devices. Beam's user-tracking service and optimizer mutate this graph at runtime for handling environment dynamics and for efficient resource usage, respectively.

Beam introduces three simple abstractions that are key to constructing and maintaining the inference graph. First, typed *inference data units* (IDUs) guide module composability. Modules can be linked to accept IDUs from other modules and generate IDUs. Second, *channels* abstract all inter-module interaction, allowing Beam to seamlessly migrate modules and mask transient disconnections when interacting modules are not collocated. Third, *coverage tags* provide a flexible and low-overhead way to connect sensors with the right coverage characteristics (e.g., location, users) to applications. We describe these key abstractions in detail next.

Inference graphs: Inference graphs are directed acyclic graphs that connect sensors to applications. The nodes in this graph correspond to *inference modules* and edges correspond to *channels* that facilitate the transmission of IDUs between modules. Figure 2 shows an example inference graph for the Quantified Self application that uses eight different devices spread across the user's home and office and includes mobile and wearable devices.

Composing an inference as a directed graph enables sharing of data-processing modules across applications and across modules that require the same input. In Beam, each compute device associated with a user, such as a tablet, phone, PC, or home hub, has a part of the runtime, called the *engine*. Engines host inference graphs and interface with other engines. Figure 3 shows two engines, one on the user's home hub and another on his phone; the inference graph for QS shown earlier is split across these engines, with the QS application itself running on a cloud server. For simplicity, we do not show other engines such as one running on the user's work PC.

IDU: An *inference data unit* (IDU) is a typed inference, and in its general form is a tuple <t,s,e>, which denotes any inference with state information *s*, generated by an inference algorithm at time *t* and error *e*. The types of the inference state *s* and error *e*, are specific to the inference at hand. An example IDU is (09/23/2015 10:10:00, occupied, 90%). Inference state *s* may be of a numerical

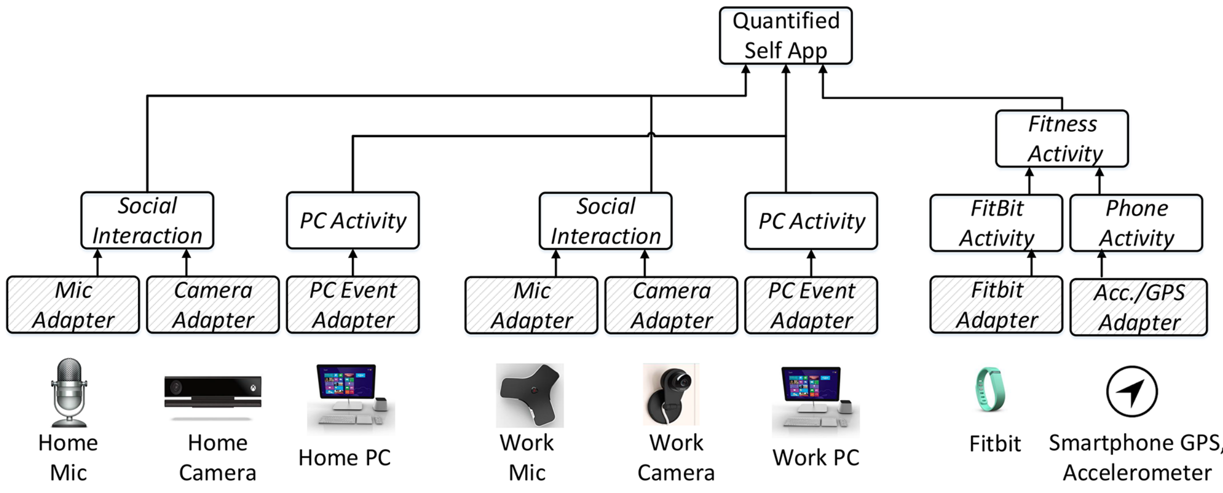


Figure 2: Inference graph for Quantified Self app

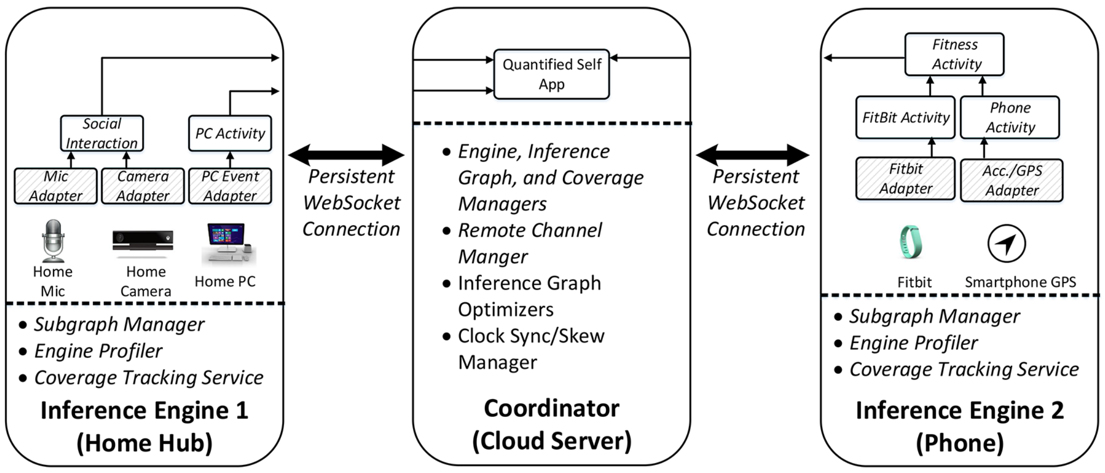


Figure 3: An overview of different components in an example Beam deployment with two engines

type such as a double (e.g., inferred energy consumption); an enumerated type such as high, medium, low; or numerical types. Similarly, error *e* may specify a confidence measure (e.g., standard deviation), probability distribution, or error margin (e.g., radius). IDUs abstract away "what is inferred" from "how it is inferred." The latter is handled by inference modules, described next.

Inference modules: Beam encapsulates inference algorithms into typed modules. Inference modules consume IDUs from one or more modules, perform certain computations using IDU data and pertinent in-memory state, and output IDUs. Special modules called *adapters* interface with underlying sensors and output sensor data as IDUs. Adapters decouple "what is sensed" from "how it is sensed." Module developers specify the IDU types a module consumes, the IDU type it generates, and the module's input dependency (e.g., {PIR} OR {camera AND mic}). Modules have complete autonomy over how and when to output an IDU and can maintain arbitrary internal state. For instance, an occupancy inference module may (1) specify input IDUs from microphone, camera, and motion sensor adapters, (2) allow multiple microphones as input, and (3) maintain internal state to model ambient noise.

Channels: To ease inference composition, *channels* link modules to each other and to applications. They encapsulate the complexities of connecting modules across different devices, including dealing with device disconnections and allowing for optimizations such as batching IDU transfers for efficiency. Every channel has a single *writer* and a single *reader* module. Modules can have multiple input and output channels. Channels connecting modules on the same engine are *local*. Channels connecting modules on two different engines, across a local or wide area network, are *remote* channels. They enable applications and inference modules to seamlessly use remote devices or remote inference modules.

Coverage tags: Coverage tags help manage sensor coverage. Each adapter is associated with a set of coverage tags that describe what the sensor is sensing. For example, a location string tag can indicate a coverage area such as "home," and a remote monitoring application can use this tag to request an occupancy inference for this coverage area. Coverage tags are strongly typed. Beam uses tag types only to differentiate tags and does not dictate tag semantics. This allows applications complete flexibility in defining new tag types. Tags are assigned to adapters at setup time using inputs from the user, and are updated at runtime to handle dynamics.

Beam's runtime also consists of a *coordinator*, which interfaces with all engines in a deployment and runs on a server that is reachable from all engines. The coordinator maintains remote channel buffers to support reader or writer disconnections (typical for mobile devices). It also provides a place to reliably store state of inference graphs at runtime while being resistant to engine crashes and disconnections. The coordinator is also used to maintain reference time across all engines. Engines interface with the coordinator using a persistent Web-socket connection, and instantiate and manage local parts of an inference graph(s).

Beam Runtime

Beam creates or updates inference graphs when applications request inferences, mutates the inference graphs appropriately to handle environmental dynamics, and optimizes resource usage.

Inference graph creation: An application may run on any user device, and the sensors required for a requested inference may be spread across devices. Applications request their local Beam

Function Application	Description

APIs:
Request(InferenceModule, List<Tag>, Mode, [SamplingRate])
Request(InferenceType, List<Tag>, Mode, [SamplingRate])
CancelRequest(InferenceModule)
	Returns a channel to specified module or to a module that outputs specified inference (and instantiates the inference graph)
	Delete channel to specified module, and terminate its inference graph

Channel APIs:
DeliverCallback(Channel, List<IDU>)
Start(), Stop()
	
	Receive a list of IDUs (invoked *on* channel reader)
	Start or stop a channel (invoked by channel reader)

Inference Module APIs:
Initialize(ModuleSpec, [SamplingRate])
PushToOutputChannels(IDU)
AllOutputChannelsStopped()
OutputChannelRestarted(Channel)
	
	Initialize the module with given specification and reporting rate
	Push inference data unit (IDU) to all output channels
	Stop sensing/processing because all output channels stopped
	Restart sensing/processing because an output channel is restarted

Optimizer APIs:
UpdateGraphs(List<Graph>, List<Engine>, App, Req/Cancel, Module, [Mode])
ReevaluateGraphs(List<Graph>, List<Engine>)
	
	Incorporates module request and returns updated list of inference graphs
	Returns updated list of inference graphs (new incarnation) after reevaluation

Table 1: Key Beam APIs: Beam offers APIs for application, inference module, and optimizer developers. Applications and inference modules use channels for communication. [] denotes an optional parameter.

engine for all inferences they require. All application requests are forwarded to the coordinator, which uses the requested inference to look up the required module. It recursively resolves all required inputs of that module (as per its specification) and reuses matching modules that are already running. The coordinator maintains a *set* of such inference graphs as an *incarnation*. The coordinator determines where each module in the inference graph should run and formulates the new incarnation. The coordinator initializes buffers for remote channels, and partitions the inference graphs into engine-specific subgraphs, which are sent to the engines.

Engines receive their respective subgraphs, compare each received subgraph to existing ones, and update them by terminating deleted channels and modules, initializing new ones, and changing channel delivery modes and module sampling rates as needed. Engines ensure that exactly one inference module of each type with a given coverage tag is created.

Inference delivery and guarantees: For each inference request, Beam returns a channel to the application. The inference request consists of (1) required inference type or module, (2) *delivery mode*, (3) coverage tags, and (4) sampling requirements (optional).

Delivery mode is a channel property that captures data transport optimizations. For instance, in the *fresh push* mode, an IDU is delivered as soon as the writer-module generates it, while in the *lazy push* mode, the reader chooses to receive IDUs in batches, thus amortizing network transfer costs from battery-limited devices. Remote channels provide IDU delivery in the face of device disconnections by using buffers at the coordinator and the writer engine. Channel readers are guaranteed (1) no duplicate IDU delivery and (2) FIFO delivery based on IDU timestamps. Currently, remote channels use the *drop-tail* policy to minimize wide-area data transfers in the event of a disconnected/lazy reader. This means that when a reader reconnects after a long disconnection, it first receives old inference values followed by more recent ones. A *drop-head* policy may be adopted to circumvent this, at the cost of increased data transfers.

When requesting inferences, applications use tags to specify coverage requirements. Furthermore, an application may specify sampling requirements as a latency value that it can tolerate in detecting the change of state for an inference (e.g., *walking* periods of more than one minute). This allows adapters and modules to temporarily halt sensing and data processing to reduce battery, network, CPU, or other resources.

Channels and modules do not persist data. Applications and modules may use a temporal datastore, such as Bolt [5], to make inferences durable.

Optimizing resource use: The Beam coordinator uses inference graphs as the basis for optimizing resource usage. The coordinator reconfigures inference graphs by remapping the engine on which each inference module runs. Optimizations are either performed *reactively* (i.e., when an application issues/cancels an inference request) or *proactively* at periodic intervals. Beam's default reactive optimization minimizes the number of remote channels, and proactive optimization minimizes the amount of data transferred over remote channels. Other potential optimizations can minimize battery, CPU, and/or memory consumption at engines.

When handling an inference request, the coordinator first incorporates the requested inference graph into the incarnation, reusing already running modules, and merging inference graphs if needed. For new modules, the coordinator decides on which engines they should run (by minimizing the number of remote channels).

Engines profile their subgraphs and report profiling data (e.g., per-channel data rate) to the coordinator periodically. The coordinator annotates the incarnation using this data and periodically reevaluates the mapping of inference modules to engines. Beam's default proactive optimization minimizes wide area data transfers.

Handling dynamics: Movement of users and devices can change the set of sensors that satisfy application requirements. For instance, consider an application that requires camera input from the device currently facing the user at any time, such as the camera on her home PC, office PC, smartphone, etc. In such scenarios, the inference graph needs to be updated dynamically. Beam updates the coverage tags to handle such dynamics. Certain tags such as those of *location* type (e.g., "home") can be assumed to be static (edited only by the user), while for certain other types, e.g., *user*, the sensed subject is mobile and hence the sensors that cover it may change.

The coordinator's *tracking service* manages the coverage tags associated with adapters on various engines. The engine's tracking service updates the user coverage tags as the user moves. For example, when the user leaves her office and arrives home, the tracking service removes the *user* tag from device adapters in the office, and adds them to adapters of devices deployed in the home. The tracking service relies on device interactions to track users. When a user interacts with a device, the tracking service appends the user's tag to the tags of all adapters on the device.

When coverage tags change (e.g., due to user movement and change in sensor coverage), the coordinator recomputes the inference graphs and sends updated subgraphs to the affected engines.

Current Prototype

Our Beam prototype is implemented as a cross-platform portable service that supports .NET v4.5, Windows Store 8.1, and Windows Phone 8.1 applications. Module binaries are currently wrapped within the service, but may also be downloaded from the coordinator on demand.

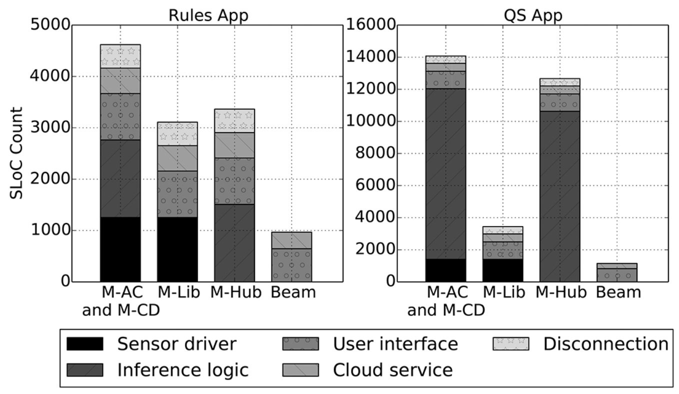


Figure 4: SLoC for different application components in the various development approaches

APIs: Table 1 shows the APIs that Beam exposes to application, inference module, and optimizer developers. Applications use the inference API to issue and cancel requests. Both inference modules and applications use the channel APIs to receive IDUs, and may *Stop* a channel to cease receiving IDUs. Each inference module is first initialized and provided with its specification and a sampling rate. It then begins its processing and pushes IDUs to all its output channels. If every output channel of a module is stopped, Beam informs the module (via AllOutputChannelsStopped), allowing it to stop its sensing/processing, thus saving resources until an output channel is restarted. Moreover, Beam abstracts optimization logic out of the coordinator, which allows modular replacement of proactive and reactive optimizers. Table 1 shows the inference graph management APIs that optimizers should implement to interface with Beam.

Inferences: We have implemented eight inference modules (mic-occupancy, camera-occupancy, appliance-use [3], occupancy, PC activity [6], fitness activity [7], semantic location, and social-interaction) and nine adapters (tablet and PC mic, powermeter, FitBit, GPS, accelerometer, PC interaction, PC event, and a HomeOS [4] adapter) to access all its device drivers.

Sample applications: We have implemented the two sample applications, *Rules* and *QS*, discussed earlier. Applications run on a cloud VM; Beam hosts the respective inference modules across the user's home PC, work PC, and phone.

Figure 4 compares the source lines of application code (SLoC) used in building these applications when using Beam and other development approaches. A monolithic approach where all sensor data is backhauled to a cloud-hosted application is denoted by *M-AC*. *M-CD* denotes an approach where a developer divides inference processing into fixed components that run on a cloud VM and end devices. *M-Lib* is similar to M-CD, except that an inference algorithm library is used. *M-Hub* denotes application development using device abstractions provided by the OS, e.g., HomeOS [4]. Moreover, we categorize the measured SLoC into the following different categories: (1) sensor drivers (one per sensor type); (2) inference algorithms, feature extraction, and learning models; (3) any required cloud-hosted services (as per the development approach) such as a storage, authentication, or access-control service; (4) mechanisms to handle device disconnections; and (5) user interface components, e.g., for displaying results or configuring devices. Using Beam results in up to 12x lower SLoC. Moreover, Beam’s handling of environmental dynamics results in up to 3x higher inference accuracy, and its dynamic optimizations match hand-optimized versions for network resource usage.

Future Directions

Our experience in building the current Beam prototype has raised interesting questions and helped us identify various directions for future work.

Beam’s current tracking service only supports tracking of users (through device interactions) and mobile devices. We aim to extend tracking support to generic objects using passive tags such as RFID or QR codes.

Similarly, we aim to enrich Beam’s optimizers to include optimizations for battery, CPU, and memory. The key challenge in doing so lies in dynamically identifying the appropriate optimization objective (e.g., network, battery), issuing reconfigurations of inference graphs, while preventing hysteresis in the system.

Many in-home devices possess actuation capabilities, such as locks, switches, cameras, and thermostats. Applications and inference modules in Beam may want to use such devices. If the inference graph for these applications is geo-distributed, timely propagation and delivery of such actuation commands to the devices becomes important and raises interesting questions of what is the safe thing to do if an actuation arrives “late.”

Lastly, by virtue of its inference-driven interface, Beam enables better information control. A user can, in theory, directly control the inferences a given application can access. In contrast, existing device abstractions only allow the user to control the flow of device data to applications, with little understanding of what information is being handed over to applications. We hope to investigate the implications of this new capability in future work.

Conclusion

Applications today are developed as monolithic silos, tightly coupled to sensing devices, and need to implement extensive data sensing and inference logic, even as devices move or have intermittent connectivity. Beam presents applications with inference-based abstractions and (1) decouples applications, inference algorithms, and devices; (2) handles environmental dynamics; and (3) optimizes resource use for data processing across devices. This approach simplifies application development, and also maximizes the utility of user-owned devices, thus surpassing current monolithic siloed approaches to building apps that use connected devices.

References

[1] IFTTT: Put the Internet to work for you: https: //ifttt.com/.

[2] The Internet of Things: http://share.cisco.com/internet-of-things.html/.

[3] N. Batra, J. Kelly, O. Parson, H. Dutta, W. J. Knottenbelt, A. Rogers, A. Singh, and M. Srivastava, “NILMTK: An Open Source Toolkit for Non-Intrusive Load Monitoring,” in *Proceedings of the 5th International Conference on Future Energy Systems*, 2014.

[4] C. Dixon, R. Mahajan, S. Agarwal, A. J. Brush, B. Lee, S. Saroiu, and P. Bahl, “An Operating System for the Home,” in *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI ’12)*, 2012.

[5] T. Gupta, R. P. Singh, A. Phanishayee, J. Jung, and R. Mahajan, “Bolt: Data Management for Connected Homes,” in *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI ’14)*, April 2014.

[6] G. Mark, S. T. Iqbal, M. Czerwinski, and P. Johns, “Bored Mondays and Focused Afternoons: The Rhythm of Attention and Online Activity in the Workplace,” in *Proceedings of the ACM CHI*, 2014.

[7] S. Reddy, M. Mun, J. Burke, D. Estrin, M. Hansen, and M. Srivastava, “Using Mobile Phones to Determine Transportation Modes,” *ACM Transactions on Sensor Networks*, vol. 6, no. 2, February 2010.

[8] B. Ur, E. McManus, M. Pak Yong Ho, and M. L. Littman, “Practical Trigger-Action Programming in the Smart Home,” in *Proceedings of the ACM CHI*, 2014.

How Kubernetes Changes Operations

BRENDAN BURNS

Brendan Burns is a Senior Staff Software Engineer at Google, Inc. and a founder of the Kubernetes project, leading engineering efforts to make the Google Cloud Platform the best place to run containers. He also has managed several other cloud teams, including Deployment Manager, Managed VMs, and Cloud DNS. Prior to Cloud, he was a lead engineer in Google's Web search infrastructure, building backends that powered social and personal search. Prior to working at Google, he was a professor at Union College in Schenectady, NY. He received a PhD in computer science from the University of Massachusetts Amherst, and a BA in computer science and studio art from Williams College. bburns@google.com

Container cluster managers are used by many Web-scale Internet companies, including Google's Borg and Omega, Facebook's Tupperware, Twitter's Aurora, and many others. At their core, these container orchestration systems schedule and manage ("orchestrate") collections of Linux application containers. In this article, I will explain the Kubernetes project.

Recently, interest in the Docker open source project has caused a significant growth in interest in Linux application containers in the general developer and operations community. Due to this growth in interest, Google launched the Kubernetes project, which makes Google's years of experience in running container clusters available to the larger world in a community-driven, open source project. The development of these internal container cluster managers was driven by real operational needs of operating software at "Google scale," but we have seen recently that their benefits apply even at a more modest scope and scale.

I illustrate how container orchestration systems change the operations tasks associated with running, maintaining, and upgrading highly scalable and reliable applications. At the heart of this change are two fundamental shifts. First, container orchestration systems provide and enforce significant decoupling between the layers of the serving stack: machine, operating system, application manager, and application code. This decoupling enables the development of specialized teams with agility and freedom to operate on their parts of the stack, thanks to separation of concerns. Second, container cluster APIs are inherently more application-oriented than traditional IaaS machine-centric APIs. This shift towards application-oriented primitives makes it easy to perform operation and maintenance tasks that were previously complicated, brittle, or both. In this article, I show how the formal boundaries introduced by containers and container cluster management enable the segmentation of traditional operations into multiple discrete roles.

In addition to a general discussion of container orchestration and operations, I also describe the Google Kubernetes container orchestrator, including the core resources in the Kubernetes system, and how they produce an inherently more stable, agile, and reliable foundation for application deployment.

Decoupling Operations Roles

Anyone who has tried to back up a trailer on a car knows that coupled, multi-component systems are hard to predict and control. Actions taken in one part of the system often cause unpredictable, user-visible problems in some other component of the system. A classic example might be upgrading a Web server, which includes updating the libc library, causing a database on the same machine to fail because the libc change introduces a bug that the database triggers.

Coupling increases the knowledge and skill set required to be a high-performing application administrator and requires operators to fully understand their entire application stack, including all dependencies, in comprehensive detail. In turn, this reduces the ability of operations teams to specialize, prevents the acquisition of true expertise, and reduces opportunities to introduce economies of scale.

As an example of this, in companies where every development team is responsible for their choice of operating system distribution (e.g., Debian or Red Hat), the operating systems in the fleet will inevitably be heterogeneous. Another example involves choosing to use SysV init vs. the systemd daemon. The resulting heterogeneity makes it difficult (if not impossible) to have a single team of administrators manage all of the machines in the fleet. It is also difficult to build a common set of tools and/or processes for performing maintenance and monitoring across all of the operating systems in the fleet. Being unable to share tooling and expertise means that fleet maintenance is more expensive and less reliable than if a single team and set of tools could manage the entire fleet of machines.

Container cluster management software makes it easier to avoid tight coupling, and the corresponding problems of heterogeneous environments, by introducing crisp boundaries and management APIs that decouple operations into discrete roles: hardware operations, kernel/OS operations, cluster operations, and application operations. The decoupling of these roles means that it is possible for each of the first three roles (hardware, kernel/OS, and cluster) to have a single team handle operations and administration, which enables lower costs and higher reliability. For application operations, it also enables the building of specialized, application-specific operations teams that can be deeply involved in the specifics of their application. The net result is a complete system that makes highly reliable applications cheaper to build and maintain.

Hardware Operations

The hardware operations role is responsible for racking and stacking machines, connecting network cables between racks and switches, and repairing or retiring machines. In modern public cloud providers, these roles have been wholly outsourced to the cloud provider, who can provide significantly greater expertise and economies of scale than the average user.

Kernel/OS Operations

The interface between a Docker container image and the underlying operating system is the Linux kernel syscall interface. Because each Docker container carries with it all of its dependencies (application binary, libraries, configuration files, etc.), it is wholly decoupled from the files that make up the machine image. An application developer can rely on two things from the kernel and operating system:

- Stability in the syscall API and operational characteristics
- A working Docker daemon

These requirements form an explicit contract between the kernel/OS and the applications that run on top of it. This means that the operations team responsible for the machine image (kernel, operating system able to boot the Docker daemon) can focus on qualifying those two generic requirements without understanding the details of any particular application. This decoupling enables release qualification, rollout, and management of a single, homogeneous kernel/OS across an entire fleet of machines. In managed container services like Google Container Engine, this kernel/operating system qualification and upgrading is outsourced to the cloud provider, enabling the application operations team (described below) to focus entirely on their application. The cluster management boundary imposes a discipline about the APIs available to application developers, as well as a single, shared implementation of this API. Because the implementation is shared between multiple, different applications, the discipline enforced by this API also acts as a counterweight to the natural tendency towards entropy and differences between the software stack supporting different applications.

Cluster Operations

If cluster users are allowed to deploy their container applications onto specific machines, then the resulting systems will be too tightly coupled because the applications will inevitably begin to rely on the specific characteristics of the machines on which they run. For example, if an application is coupled to the machine's network identity (hostname and IP address), the decoupling between application, hardware, and kernel has been broken. That machine cannot just be sent to repairs when the hardware operator determines it is failing. Nor can it be rebooted for an OS installation any time the OS operator decides one is needed. It is the container cluster manager's goal to decouple containerized applications from the specifics of any particular machine. For example, in Kubernetes, we give each pod an IP address that is independent of the IP address of the machine that it is running on. The pod does not have access to the machine's network identity. Furthermore, Kubernetes can restrict the set of file systems that can be mounted into a pod from the host file system, restricting access to things like raw block devices and other machine-specific hardware.

Additionally, container cluster managers, like Kubernetes, provide a declarative, programmable API that is the primary one by which developers schedule and deploy users' applications onto a fleet of machines. Consequently, developers are decoupled from the details of physical machines, because their mode of interaction is container and application-centric. The particular details of the machine that ends up running the application developer's containers become an implementation detail of the underlying cluster manager.

Indeed, many users forget that their applications are running on physical (or virtual) machines at all and, instead, deal solely with the logical compute substrate provided by the container cluster API. They ask that API for a certain set of application resource requirements (say, two cores and 100 GB of RAM), and it is the container cluster manager's responsibility to find sufficient resources somewhere in the cluster and deploy the application onto those resources. The job of a container cluster administrator is to ensure that the services that provide the container cluster management API stay available and operationally healthy at all times.

Application Operations

Closest to the end user in these decoupled operations roles are application operators. These administrators are focused on managing and deploying applications: for example, the Google search backend or Gmail frontend. These administrators develop deep specialized knowledge of their applications, and rely on cluster, kernel, and hardware operations teams to provide them the infrastructure they need to do their job. Transferring work that is unrelated to their application (e.g., kernel and OS upgrades) onto specialized kernel operation teams allows the application operation teams to develop application-specific tooling for more reliable management of their application. The specialization of application administrators on a particular application also means that they can develop deep technical understanding of the specific application software, and form significant partnerships with the development teams to improve the reliability and performance of that software.

The Kubernetes Cluster Manager

Having described how containers and cluster management APIs enable the decoupling of operations roles, I will now discuss some specifics of the Kubernetes API to provide a deeper understanding of the functionality that Kubernetes provides. Beginning with a description of pods, the atomic unit of scheduling in the Kubernetes system and the basic building block for running containers in a Kubernetes cluster, I will go on to cover generic software patterns for building applications with pods. I'll show how Kubernetes resources are organized into dynamic sets with labels and how those labels are used to automatically manage replicated microservices using Replication Controllers and Services.

Pods

Pods are the most fundamental API object in Kubernetes. A pod is a group of containers that is scheduled together onto one machine. All of the containers within a pod share the same network namespace, so the containers within a pod can easily find each other on "localhost." This eliminates the need for a complicated discovery service (more on that later). The containers in a pod also share the same IPC namespace, which means that they can use traditional UNIX IPC, such as pipes. As Kubernetes matures, we expect that pods will come to share all of the available kernel namespaces, including group ID namespaces, process ID namespaces, and more.

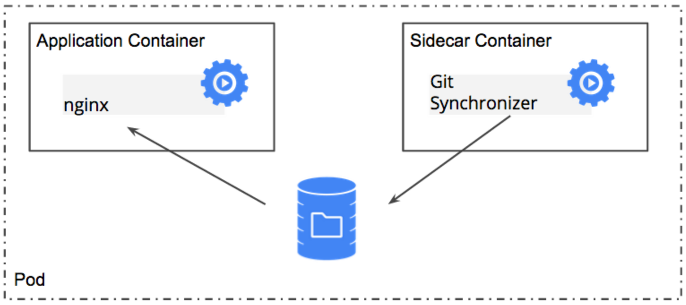


Figure 1: Example of a sidecar container: a pod where an Nginx Web server is being augmented by a Git synchronizing container

Pods also encapsulate node-level health checking and reliability for their constituent containers. In Kubernetes, there are two different types of checks:

First, each container has a *liveness* check. By default, this is a simple process-based one ("is the process running"), but it can be extended to include several other application-specific health checks: *HTTP* (healthy if the container endpoint returns an HTTP 200), *TCP* (healthy if a TCP socket can be opened), or *exec* (healthy if a user-supplied binary executed in the context of the container returns an exit code of zero). If any liveness test fails, the container is automatically restarted by Kubernetes.

The second check is a *readiness* check, which is applied to an entire pod. Readiness checks indicate whether the pod is ready to serve end-user traffic. In many situations, a pod may take some time to start up, due to network downloads, migrations, or other long, computational initialization steps. During this time, the pod is *alive*: it should not be restarted by Kubernetes. However, it is not *ready*: it should not serve traffic. Readiness checks are used to implement service load balancers, described below.

Pod Patterns

When you start using pods, some general patterns naturally start to recur. The three common ones are sidecar containers, ambassador containers, and adapter containers.

SIDECAR CONTAINERS

Sidecar containers extend and enhance the "main" container; they take existing containers and make them better.

As an example, consider a container that runs the Nginx Web server. Add a different container that syncs a directory with a Git repository, share the file system between the containers, and you have built a non-atomic, push-to-deploy Git. But you've done it in a modular manner where the Git synchronizer can be built by a different team and reused across many different Web servers (Apache, Python, Tomcat, etc.). Because of this modularity, you only have to write and test your Git synchronizer once to reuse it across numerous apps. If someone else writes it, you don't even need to do that.

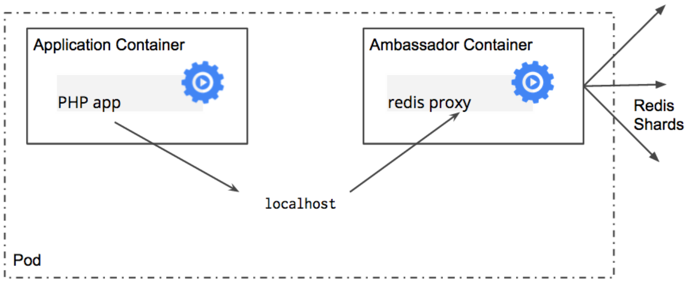


Figure 2: Example ambassador container: a pod where a Redis proxy ambassador is used to proxy connections from a PHP application to a set of Redis shards

AMBASSADOR CONTAINERS

Ambassador containers proxy the outside world via a local connection in the same pod.

As an example, consider a Redis cluster with read replicas and a single write master. You can create a pod that groups your Redis client with a Redis ambassador container. The ambassador is a proxy; it is responsible for splitting reads and writes to Redis and sending them on to the appropriate Redis servers. Because these two containers share a network namespace, they share an IP address, and your application can open a connection on "localhost" and find the proxy without any service discovery. Note that this is "localhost" for the network of the pod, *not* "localhost" on the host machine.

ADAPTER CONTAINERS

Adapter containers standardize and normalize output.

In any real-world application, the application's software comes from a heterogeneous set of sources (open source, off-the-shelf software, home brew), and monitoring system developers cannot be expected to understand, build, maintain, and deploy for all of them. Consequently, you often need to wrap applications to enable communication with auditing or monitoring services.

Using a modular *adapter* container co-located in the same pod as your application gives you a simple unit of deployment that combines both application and adapter. Using adapters enables each application developer to supply a common interface. The modularity of using two different containers (the application and the adapter) means that despite making the adapter the application owner's responsibility, adapters can be reused (e.g., a Java JMX adapter).

The adapter pattern creates pods that group the application containers with adapters that know how to do the transformation. Again, because these pods share namespaces and file systems, the coordination of these two containers is simple and straightforward.

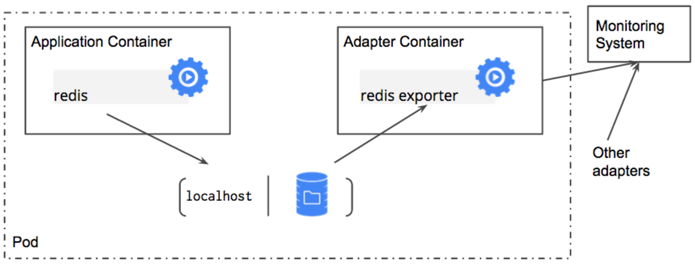


Figure 3: Example of an adapter container: a pod where the Redis key-value store is adapted to provide a consistent monitoring interface (e.g., https://github.com/oliver006/redis_exporter)

Labels

Experience operating large, complicated systems has taught us that requiring applications and their parts to be grouped into fixed, disjoint sets is overly restrictive.

As an example of this, consider the canonical search stack. There is a set of replicas that are responsible for serving end-user requests {frontend, middleware, backend servers}, and then there are the jobs that are responsible for building, pushing, and loading a new search index {crawler, index-builder, backend servers}. The presence of "backend servers" in both of these organizations reflects the problem with fixed sets. We need an organizational mechanism that can flexibly represent both of these organizational sets (and any other useful sets). If the cluster management infrastructure can't represent the overlapping sets of organizations that are present in the cluster, then additional tooling, which is opaque to the cluster manager, will get built to represent these organizations. The additional complexity required to make these systems interact well with the cluster management software makes the system harder to maintain and extend.

Additionally, we need a representation that is dynamic. For example, at different times, pods may be added or removed from sets; during a rolling update of a service to a new version of its software, pods are dynamically added and removed from the set of backends of a load balancer. We need a representation that can easily capture this dynamism without requiring constant action from the user to maintain these sets.

In Kubernetes, *labels* and *label queries* provide flexible, dynamic sets of resources. Rather than encode any specific grouping primitive into the Kubernetes API, every resource in the Kubernetes API can have labels attached to that resource. These labels are arbitrary, key-value pairs that help define the object. For example, a production Web server might have the labels {`role=frontend, stage=production, version=v1, machine=m1, rack=r2`}, and a production backend might have the labels {`role=backend, stage=production, version=v1, machine=m2, rack=r1`}.

A label query dynamically organizes objects into a group by constructing a set of objects that match its conditions. For example, we might query "`stage=production`" to see all production pods, "`rack=r2`" to see all containers on a particular rack, or even conjugate queries like "`stage=production, machine=m1`" for all production jobs on a particular machine. Label queries are used to list particular RESTful resources in the Kubernetes API. A label query for a resource of a particular type (e.g., pods) will only return the pods whose labels match the query.

Reconciliation

The third key concept in Kubernetes, after pods and labels, is reconciliation loops.

The basic premise is that there are three states of the world: an idealized *desired state*, which is a declarative statement of what the world should be like; a *current state*, which approximates the actual state, and might be noisy, incomplete, or out of date; and an *actual state*. Unfortunately, the actual state isn't directly observable, thanks to the vagaries of distributed systems, delays, and failures, so we must make do with the observed state.

The role of the reconciliation loop is to repeatedly compare the current state against the desired state, and take action to drive the actual state to match the desired state. This is just a control loop, like the one in your thermostat. It is what transforms Kubernetes into a self-healing, dynamic system, by automatically causing it to restore the system to the desired state without needing operator intervention. Only if this fails does the system need to invoke help from an administrator, e.g., by triggering an alert.

Replication Controllers

In any real production system, replicating the components in the system is the only way to achieve reliable operation. Each replica is an independent unit of failure, and thus, multiple replicas reduce the probability of a total failure. They also allow a service to be scaled up as traffic grows. However, the complexity of managing a replicated system must not be linear in the number of replicas, or else the system is not truly scalable.

In Kubernetes, replication controllers provide an API for managing replicated sets of pods. Replication controllers use a pod template, a label query, and a desired number of replicas to create a replicated set of pods. The operation is as follows:

```
Repeat forever
  1. Select pods matching Label Query.
  2. Subtract number of pods found from the desired number of
replicas.
  3. If this difference is negative, destroy a pod.
  4. If this difference is positive, create a pod using the pod
Template.
```

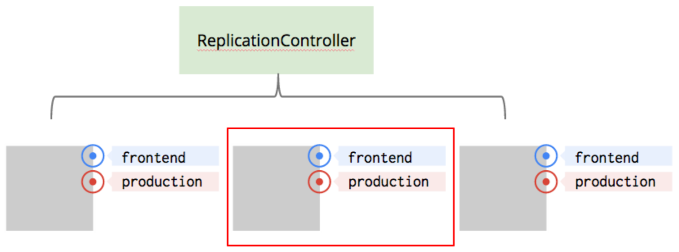


Figure 4. A replicated set of pods with a misbehaving replica (pod within rectangle). Solid boxes are pods; circles indicate labels attached to them.

Note that this is a reconciliation loop. No matter why a pod disappears—whether due to node failure, accidental deletion, or network partition—the replication controller attempts to ensure that the correct number of replicas exists. Likewise, if a user or automated process resizes the number of replicas up or down, these adjustments to the number of replicas are also materialized by this simple reconciliation loop.

Services

A recent, popular trend in distributed systems is microservice architectures, which decouple different pieces of a distributed system into independently managed and scaled microservices. This decoupling helps microservice architectures to be reliable and scalable.

In Kubernetes, the *Service* API object represents a load balancer for a microservice. Like replication controllers, services are based on a dynamic label query that identifies the set of backends that the service connects to.

To enable service discovery, a service is assigned a static virtual IP address (VIP). This address is constant, and has the same lifespan as the service. Consequently, the VIP can be populated into DNS for service discovery. Because the VIP is not the address of any particular pod, the VIP can be kept constant, even as pods are scaled up or down behind the service.

Kubernetes itself ships with a simple, default load-balancer implementation, but the Kubernetes API also makes *Endpoint* objects available. These endpoints are the current members of the service's load balancing group—i.e., the pod IP addresses across which it spreads incoming requests. Advanced users can use these endpoints to populate a third-party load balancer (e.g., Nginx, HAProxy) or even to implement thick clients that do balancing without a proxy.

The maintenance of the service's endpoints is another example of a reconciliation loop. In this case, the loop looks like:

```
Repeat forever
  1. Select pods matching Service Selector Label Query
  2. For each matching pod
    a. If the pod is *Ready* (see 'Readiness Checks' above)
      i. Add the pod to the Endpoint set for this Service
```

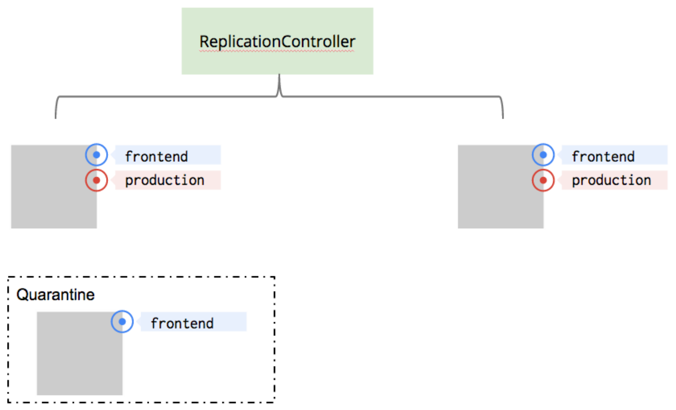


Figure 5. After the "production" label is removed from the misbehaving replica, the replica is now quarantined.

Operations in Kubernetes

It is easier to operate systems that are deployed into a Kubernetes cluster than systems deployed into traditional virtual machines. This section describes two operations scenarios that demonstrate this.

Quarantining a Replica

One of the common tasks that occur in operations is quarantining a misbehaving replica of an application. Oftentimes, sadly, this means simply killing the misbehaving replica, collecting logs for retrospective analysis, and restarting the process. While this restores the service to health quickly, it is much harder to debug a problem from (possibly incomplete) logs than it is with a running server. It would be better to remove a misbehaving replica from the service but maintain it as a running server so that it can be debugged. This is precisely what Kubernetes services and labels allow. This is illustrated in the following example.

We start with an existing Kubernetes replicated service that shares load across three pods. The pod in the middle is determined to be misbehaving.

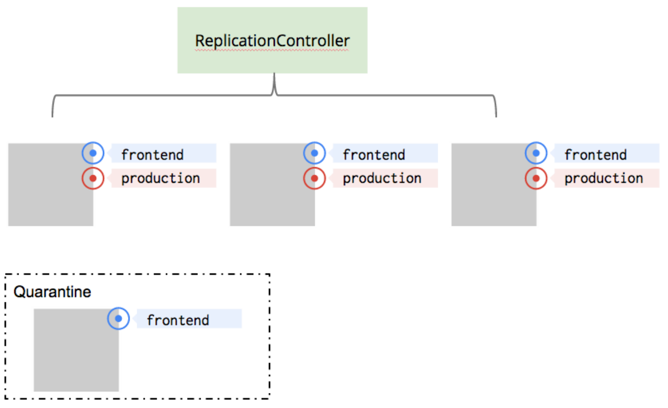


Figure 6. The replication controller replaces the misbehaving pod with a new replica.

The operator removes the "production" label from the misbehaving pod. Because Kubernetes dynamically queries label selectors, the pod is now removed from the corresponding ReplicationController and the service.

The reconciliation loop in `ReplicationController` detects that a pod is missing from the replica set and creates a new pod, restoring the service to full health. The misbehaving pod is retained for future debugging.

Rolling Update

Another common operation is rolling out new software. Kubernetes achieves this through manipulating replication controllers and labels.

At the start of the update, there is a single replication controller. It has three replicas, and is using version 1 (v1) of the application. There is also a Kubernetes Service that is defined to serve traffic to pods with the "frontend" and "production" labels. To perform a rolling update, a second replication controller is created. This replication controller is identical to the first replication controller in all ways, except the image in its template has been updated to version 2 (v2). Initially, the desired replica count for this controller is set to zero (Figure 7).

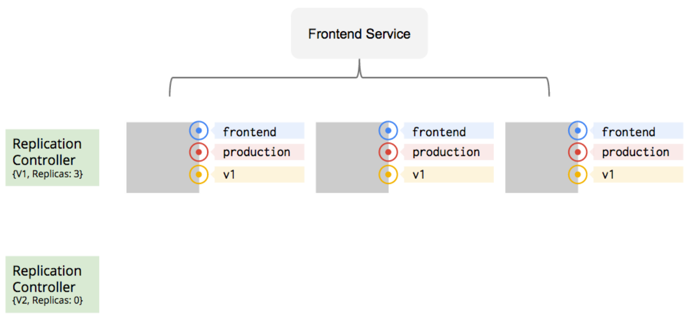

Frontend Service
Replication Controller {V1, Replicas: 3}
frontend
production
v1
frontend
production
v1
frontend
production
v1
Replication Controller {V2, Replicas: 0}

Figure 7. The initial state of the rolling update. A second replica controller has been created but has no replicas yet.

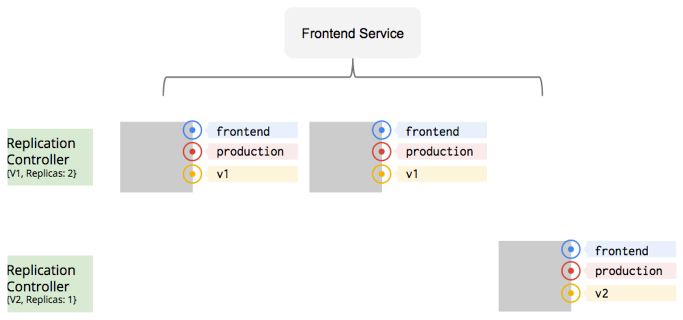

Frontend Service
Replication Controller {V1, Replicas: 2}
frontend
production
v1
frontend
production
v1
Replication Controller {V2, Replicas: 1}
frontend
production
v2

Figure 8. The first "canarying" step of a rolling update: replica count for the original controller is set to two, and to one for the second controller.

To perform the rolling update, the desired number of replicas on the v1 replication controller is dialed down by one (in this case, to two replicas), and the desired replicas for the v2 replication controller is increased by one (to one replica, Figure 8).

This process of one up, one down proceeds until the desired number of replicas for v1 is zero and the desired number of replicas for v2 is three. Because the Kubernetes Service is defined by the label query `{role=frontend, stage=production}`, which ignores the version, the load balancer seamlessly spreads traffic across version 1 and version 2 as the rollout proceeds. If failures occur during the rollout, and a rollback is necessary, it is simple to reverse the roles of the replication controllers and restore the number of replicas for v1 to be three.

Conclusion

Containers have grown in popularity because they decouple user applications from the underlying operating system/kernel, and allow the development of kernel/OS-specific operations teams. Container cluster orchestration systems, like Kubernetes, further allow the decoupling of operations into hardware operations, kernel operations, cluster operations, and application operations. This decoupling enables specialization and focus, which increases the reliability and scalability of those operations teams. Furthermore, Kubernetes provides a set of objects that makes it easier for application developers to design and develop services that are easier to operate and scale. Container cluster management systems are the backbone of most large-scale Web service companies, and with the advent of open source solutions like Docker and Kubernetes, we believe there is an industry-wide shift underway to this new style of decoupled infrastructure.

Acknowledgments
I would like to thank John Wilkes, Jessie Yang, Robert van Gent, and Seth Hettich for providing significant feedback and revisions to this article.

Being an On-Call Engineer
A Google SRE Perspective

ANDREA SPADACCINI AND KAVITA GULIANI

Andrea Spadaccini works in Dublin as a Site Reliability Manager for Google, which he joined in 2012 as an SRE working on the systems that distill, store, and serve all the metrics about Google's Ads platforms. Prior to that, he worked on Linux-based PBX products, hacked on open source CPU simulators, and co-founded a nonprofit for students to get work experience while pursuing their studies. He earned a PhD in computer engineering from the University of Catania, Italy, where he focused mostly on biometric recognition. spadaccio@google.com.

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

Being on-call is a critical duty that many operations and engineering teams must undertake in order to keep their services reliable and available. However, there are several pitfalls in the organization of on-call rotations and responsibilities that can lead to serious consequences for the services and for the teams if not avoided. We provide the primary tenets of the approach to on-call that Google's Site Reliability Engineers have developed over years, and explain how that approach has led to reliable services and sustainable workload over time.

Several professions require employees to perform some sort of on-call duty, which entails being available for calls during both working and non-working hours. In the IT context, on-call activities have historically been performed by dedicated Ops teams tasked with the primary responsibility of keeping the service(s) for which they are responsible in good health.

Many important services in Google, e.g., Search, Ads, and Gmail, have dedicated teams of Site Reliability Engineers (SREs) [1] responsible for the performance and reliability of these services. As such, SREs are on-call for the services they support. The SRE teams are quite different from purely operational teams in that they place heavy emphasis on the use of engineering to approach problems. These problems, which typically fall in the operational domain, exist at a scale that would be intractable without software engineering solutions.

To enforce this type of problem-solving, Google hires people with a diverse background in systems and software engineering into SRE teams. We cap the amount of time SREs spend on purely operational work at 50%; at minimum, 50% of an SRE's time should be allocated to engineering projects that further scale the impact of the team through automation, in addition to improving the service.

We present an informed view of how Google SRE teams organize the on-call aspect of their jobs, and how Google's strong focus on engineering determines numerous aspects of this organization.

We do not describe all the possible ways of organizing on-call rotations in detail. For detailed analysis, refer to the "Oncall" chapter of *The Practice of Cloud System Administration* [2].

Life of an On-Call Engineer

As the guardian of production systems, the on-call engineer takes care of his or her assigned operations by managing outages that affect the team and performing and/or vetting production changes.

When on-call, an engineer is available to perform operations on production systems within minutes, according to the paging response Service Level Objectives (SLOs) agreed to by the team and the business system owners. Typical SLO values are five minutes for user-facing or otherwise highly time-critical services, and 30 minutes for less time-sensitive systems. The company provides the page-receiving device, which is typically a phone. Google has flexible alert delivery systems that dispatch pages via multiple mechanisms (email, SMS, robot call, app) across multiple devices.

This page-to-work-towards-resolution SLO is distinct from the service SLOs themselves (e.g., user-facing latency, processing delay, and so on). There is a relationship between the two types of SLOs: the service SLOs imply upper bounds for the page-to-work-towards-resolution SLO. For example, if a user-facing system must obtain 4 nines of availability in a given quarter (99.99%), the allowed quarterly downtime is around 13 minutes. This constraint implies that the reaction time of on-call engineers has to be on the order of minutes. For systems with more relaxed SLOs, the reaction time can be on the order of tens of minutes.

As soon as a page is received and acknowledged, the on-call engineer is expected to triage the problem and work towards its resolution, possibly involving other team members and escalating as needed.

Non-paging production events, such as lower priority alerts or software releases, can also be handled and/or vetted by the on-call engineer during business hours. These activities are less urgent than paging events, which take priority over almost every other task, including project work.

Many teams have both a primary and a secondary on-call rotation. The distribution of duties between the primary and the secondary varies from team to team and ranges from the secondary acting as a fall-through for the pages missed by the primary on-call to an arrangement in which the primary on-call handles only pages and the secondary handles all other non-urgent production activities.

In teams for which a secondary rotation is not strictly required for duty distribution, it is common for two related teams to serve as secondary on-call for each other, with fall-through handling duties. This setup eliminates the need for an exclusive secondary on-call rotation.

Balanced On-Call

SRE teams have specific constraints on the quantity and quality of on-call shifts. The quantity of on-call can be calculated by the percentage of time spent by engineers on on-call duties. The quality of on-call can be calculated by the number of incidents that occur during an on-call shift.

SRE managers are responsible for keeping the on-call workload balanced and sustainable across these two axes.

Balance in Quantity

SREs can spend no more than 25% of their time on-call, and another 25% of their time on other types of operational, non-project work. We strongly believe that the “E” in “SRE” is a defining characteristic of our organization, so we strive to invest at least 50% of SRE time in engineering.

Using the 25% rule, we can derive the minimum number of SREs required to sustain a 24/7 on-call rotation. Assuming that there are always two people on-call (primary and secondary, with different duties), the minimum number of engineers needed for on-call duty from a single-site team is eight: assuming week-long shifts, each engineer is on-call (primary or secondary) for one week every month. For dual-site teams, a reasonable minimum size of each team is six, both to honor the 25% rule and to ensure a substantial and critical mass of engineers for the team.

If a service implies enough work to justify growing a single-site team, we can create a multi-site team. A multi-site team can be advantageous for two reasons:

- Night shifts have detrimental effects on people’s health [3], and multi-site rotation allows teams to avoid night shifts altogether.
- Limiting the number of engineers in the on-call rotation ensures that engineers do not lose touch with the production systems (see “A Treacherous Enemy: Operation Underload,” below).

However, multi-site teams incur communication and coordination overhead. Therefore, the decision to go multi-site or single-site should be based on the tradeoffs each option entails, the importance of the system, and the workload each system generates.

Balance in Quality

For each on-call shift, an engineer should have sufficient time to deal with incidents and follow-up activities such as writing postmortems [4]. Assuming that on-call incidents, on average, require six hours of work between investigation, root cause analysis, remediation, and follow-up activities such as writing a postmortem, it follows that the maximum number of incidents per day is two. In order to stay within this upper bound, the distribution of paging events should be very flat over time, with a likely median value of 0: if a given component or issue causes pages every day (median incidents/day 1), it is likely that something else will break at some point, thus causing more incidents than should be permitted.

If this limit is temporarily exceeded, e.g., for a quarter, corrective measures should be put in place to make sure that the operational load returns to a sustainable state (see “Avoiding Operational Overload,” below).

Compensation
Adequate compensation needs to be considered for out-of-hours support. Different organizations handle on-call compensation in different ways; Google offers time-off-in-lieu or straight cash compensation, capped at some proportion of overall salary. The compensation cap represents, in practice, a limit on the amount of on-call work that will be taken on by any individual. This compensation structure ensures incentivization to be involved in on-call duties as required by the team, but also promotes a balanced on-call work distribution and limits potential drawbacks of excessive on-call work, such as burnout or inadequate time for project work.

Feeling Safe

As mentioned earlier, SRE teams support Google's most critical systems. Being an SRE on-call typically means assuming responsibility for user-facing, revenue-critical systems, or for the infrastructure required to keep these systems up and running. SRE methodology for thinking about and tackling problems is vital for the appropriate operation of services.

Modern research identifies two distinct ways of thinking that an individual may choose, consciously or subconsciously, when faced with challenges:

- Intuitive, automatic, and rapid action
- Rational, focused, and deliberate cognitive functions [5]

When dealing with the outages related to complex systems, the second of these options is more likely to produce better results and lead to well-planned incident handling.

To make sure that the engineers are in the appropriate frame of mind to leverage the latter mindset, it's important to reduce the stress related to being on-call. The importance and the impact of the services and the consequences of potential outages can create significant pressure on the on-call engineers, damaging the well-being of individual team members and possibly prompting SREs to make incorrect choices that can endanger the availability of the service. Stress hormones like cortisol and CRH are known to cause behavioral consequences—including fear—that can impair cognitive functions and cause suboptimal decision-making [6].

Under the influence of these stress hormones, the more deliberate cognitive approach is typically subsumed by unreflective and unconsidered (but immediate) action, leading to potential abuse of heuristics. Heuristics are very tempting behaviors when on-call. For example, when the same alert pages for the fourth time in the week, and the previous three pages were initiated by an external infrastructure system, it is extremely tempting to exercise confirmation bias by automatically associating this fourth occurrence of the problem with the previous cause.

While intuition and quick reactions can seem like desirable traits in the middle of incident management, they have downsides. Intuition can be wrong and is often less supportable by obvious data. Thus, following intuition can lead an engineer to waste time pursuing a line of reasoning that is incorrect from the start. Quick reactions are deep-rooted in habit, and habitual responses are unconsidered, which means they can be disastrous. The ideal methodology in incident management strikes the perfect balance between taking steps at the desired pace when enough data is available to make a reasonable decision and simultaneously critically examining your assumptions.

It's important that on-call SREs understand that they can rely on several resources that make the experience of being on-call less daunting than it may seem. The most important on-call resources are:

- Clear escalation paths
- Well-defined incident-management procedures
- A blameless postmortem culture [4]

The developer teams of SRE-supported systems usually participate in a 24/7 on-call rotation, and it is always possible to escalate to these partner teams when necessary. The appropriate escalation of outages is generally a principled way to react to serious outages with significant unknown dimensions.

When handling incidents, if the issue is complex enough to involve multiple teams, or if, after some investigation, it is not yet possible to estimate an upper bound for the incident's time span, it can be useful to adopt a formal incident-management protocol. Google SRE uses the protocol described in "Managing Incidents" [7], which offers an easy to follow and well-defined set of steps that aid an on-call engineer in rationally pursuing a satisfactory incident resolution with all the required help. This protocol is internally supported by a Web-based tool that automates most of the incident management actions, such as handing off roles and recording and communicating status updates. This tool allows incident managers to focus on dealing with the incident, rather than spending time and cognitive effort on mundane actions such as formatting emails or updating several communication channels at once.

Finally, when an incident occurs, it's important to evaluate what went wrong, recognize what went well, and take action to prevent the same errors from recurring in the future. SRE teams must write postmortems after significant incidents, and detail a full timeline of the events that occurred. By focusing on events rather than the people, these postmortems provide significant value. Rather than placing blame on individuals, value is derived from the systematic analysis of production incidents. Mistakes happen, and software should make sure that we make as few mistakes as possible. Recognizing automation opportunities is one of the best ways to prevent human errors [4].

Avoiding Inappropriate Operational Load

Operational Overload

As mentioned in the "Balanced On-Call" section above, SREs spend at most 50% of their time on operational work. What happens if operational activities exceed this limit? The SRE team and leadership are responsible for including concrete objectives in quarterly work planning in order to make sure that the workload returns to sustainable levels.

Ideally, symptoms of operational overload should be measurable, so that goals can be quantified (e.g., number of daily tickets < 5, paging events per shift < 2).

Monitoring misconfiguration is a common cause of operational overload. Paging alerts should be aligned with the symptoms that threaten a service's SLOs. All paging alerts should also be actionable. Low-priority alerts that bother the on-call engineer every hour (or more frequently) disrupt productivity, and the fatigue such alerts induce can also cause serious alerts to be treated with less attention than necessary.

It is also important to control the number of alerts that the on-call engineers receive for a single incident. Sometimes a single abnormal condition can generate several alerts, so it's important to regulate the alert fanout by ensuring that related alerts are grouped together by the monitoring or alerting system. If, for any reason, duplicate or uninformative alerts are generated during an incident, silencing those alerts can provide the necessary quiet for the on-call engineer to focus on the incident itself. Noisy alerts that systematically generate more than one alert per incident should be tweaked to approach a 1:1 alert/incident ratio. Doing so allows the on-call engineer to focus on the incident instead of triaging duplicate alerts.

Sometimes the changes that cause operational overload are not under the control of the SRE teams. For example, the application developers might introduce changes that cause the system to be more noisy, less reliable, or both. In this case, it is appropriate to work together with the application developers to set common goals to improve the system.

In extreme cases, SRE teams may have the option to "give back the pager"—SRE can ask the developer team to be exclusively on-call for the system until it meets the standards of the SRE team in question. Giving back the pager doesn't happen very frequently, as it's almost always possible to work with the developer team to reduce the operational load and make a given system more reliable. In some cases, though, complex or architectural changes spanning multiple quarters might be required to make a system sustainable from an operational point of view. In such cases, the SRE team should not be subject to an excessive operational load. Instead, it is appropriate to negotiate the reorganization of on-call responsibilities with the development team, possibly routing some or all paging alerts to the developer on-call. Such a solution is typically a temporary measure, during which time the SRE and developer teams work together to get the service in shape to be onboarded by the SRE team again.

The possibility of renegotiating on-call responsibilities between SRE and developer teams attests to the balance of powers between the teams. This working relationship also exemplifies how the healthy tension between these two teams and the values that they represent—reliability vs. feature velocity—is typically resolved by greatly benefitting the service and, by extension, the company as a whole.

A Treacherous Enemy: Operation Underload

Being on-call for a quiet system is blissful, but what happens if the system is too quiet or when SREs are not on-call often enough? An operation underload is undesirable for an SRE team. Being out of touch with production for long periods of time can lead to confidence issues, both in terms of overconfidence and underconfidence, while knowledge gaps are discovered only when an incident occurs.

To counteract this eventuality, SRE teams should be sized to allow every engineer to be on-call once or twice a month, thus ensuring that each team member is sufficiently exposed to production.

Some teams also run so-called "Wheel of Misfortune" exercises, in which theoretical (or practical) incident scenarios are presented to the team by a dungeon master, much in the style of traditional role-playing games. This exercise is also a useful team activity that can help to hone and improve troubleshooting skills and knowledge of the service.

Google also has a company-wide annual disaster recovery event called DiRT (Disaster Recovery Training) that combines theoretical and practical drills to perform multi-day testing of infrastructure systems and individual services.

Onboarding New Systems

It is common for SRE teams to become responsible for new systems, a process that typically culminates in handing off pager responsibilities, also called onboarding.

The SRE team needs to engage with the new system well before the onboarding process starts. Ideally, the SREs are involved from the early design phase of the new system, as their knowledge and experience with the production infrastructure can offer an important perspective on the architecture of the new systems. Direct involvement by SREs during the development phase might be necessary as the system approaches its launch, in preparation for a Production Readiness Review (PRR) or Launch Review.

After the new system launches, the application developers may remain on-call for the system until the ownership is transitioned to SRE. A system must meet specific requirements with regards to reliability, Service Level Objectives (SLOs), alerting, and the on-call load before it is onboarded by SRE. The on-call training can begin towards the end of the onboarding process. Generally, the application developers train SREs on the internals of the new systems, explaining the most likely or common failure modes and how to react to these failures. To demonstrate debugging techniques, developers may fake troubleshooting scenarios and demonstrate their resolution to SREs.

All alerts are expected to have corresponding documentation that enables the on-call engineer to take appropriate actions when paged. Upon service handoff, documentation ownership is transitioned to SREs, who are expected to keep the docs up-to-date in collaboration with the application developers.

Conclusion

The approach to on-call we described serves as a guideline for all SRE teams in Google and is key to fostering a sustainable and safe work environment. Google's approach to on-call has enabled us to use engineering work as the primary means to scale production responsibilities and maintain high reliability and availability despite the increasing complexity and number of systems and services for which SREs are responsible.

References

[1] http://www.site-reliability-engineering.info/.

[2] Thomas A. Limoncelli, Strata R. Chalup, Christina J. Hogan, "Oncall," in *The Practice of Cloud System Administration: Designing and Operating Large Distributed Systems*, vol. 2, Pearson Education, 2014.

[3] Jeffrey S. Durmer and David F. Dinges, "Neurocognitive Consequences of Sleep Deprivation," in *Seminars in Neurology*, vol. 25, no. 1, 2005.

[4] Jake Loomis, "How to Make Failure Beautiful: The Art and Science of Postmortems," in *Web Operations: Keeping the Data on Time*, O'Reilly Media, 2010.

[5] Daniel Kahneman, *Thinking, Fast and Slow,* Farrar, Straus and Giroux, 2011.

[6] George P. Chrousous, "Stress and Disorders of the Stress System," *Nature Reviews Endocrinology,* vol. 5, July 2009, doi: 10.1038/nrendo.2009.106.

[7] Andrew Stribblehill, Kavita Guliani, "Managing Incidents," *;login:,* vol. 40, no. 2, April 2015: https://www.usenix.org/publications/login/apr15/stribblehill.

/var/log/manager

How Technical Managers Tell Time

ANDY SEELY

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

Time management for sysadmins is a largely solved equation thanks to Mr. Limoncelli [1]. I would like to offer a humble extension to his work and talk about time management for the technical manager.

Normal and Interrupt for the Sysadmin

The sysadmin has recurring tasks. Flush logs. Check backups. Monitor loads. Look up *Simpsons* quotes to use in the next change control meeting. Answer email from the manager. Probably in that order. Managers know they rate below looking up pop culture references.

The sysadmin has interrupt-driven tasks that trump all the recurring, normal tasks. The prioritization is now whatever the interrupt signal is. The datacenter is on fire. The SAN just crashed. The boss's printer is out of paper. The public-facing e-commerce site certificate expired. You know, the critical break-fix things that are instantly more important than anything that may have been planned out in advance.

Normal and Interrupt for the Technical Manager

The technical manager has recurring tasks. Read and answer email. Listen to and answer voice mail. Check and update calendars. Attend scheduled meetings. Meet weekly deadlines like time card queues. Prepare reports and briefings. Take any administrative actions required, like approving expenses, denying training requests (don't be disappointed, you may resubmit again in 30 days for further denial!), and responding to requests for information from the VP.

The technical manager has interrupt-driven events that just move the recurring tasks to later in the day. The VP overheard something in the board meeting and wants an explanation. Another VP wants to talk about his golf swing and you're the first person he sees. Another manager wants to complain about your people doing something wrong. A customer wants to know when a project will be delivered, with a full review of schedule, today. HR has to have a meeting immediately to discuss a complaint someone filed. An employee is in a bind and needs top-cover.

Taking Control of the Manager's Information Flow

It's all true, and it all has to get done. There are some practical tips and tricks I can offer to help a busy technical manager never forget a promise (or a threat) and always have the right answers. There are two things you need above everything else, even above a vacation where you take your laptop and work anyway: discipline and a system. I'm going to tell you about my system.

First, you have to baseline. Understand your own inputs. For me, my inputs span multiple inject points that I cannot coalesce any further for a variety of reasons:

1. Multiple calendars
2. Multiple inboxes
3. Two telephones with voice mail
4. Drive-by tasks and requests for information from my peers and leadership
5. Drive-by information updates and status reports from my team
6. Drive-by requests for close air support when one of my team needs me to help them
7. Scheduled meetings
8. Unscheduled meetings
9. The hallway, which is where a surprising amount of coordination seems to get done

My durable repositories of information:

1. My active inboxes
2. My email archives
3. SharePoint portal "wiki" file
4. My analog, handwritten, completely illegible notebook

After you've articulated what your inputs and repositories are, you need to have a system for processing flow. My system owes a debt to the Getting Things Done approach [2].

First, own your inbox and make it work for you. Create at least seven folders in your inbox. Label them Monday, Tuesday, Wednesday, Thursday, Friday, Sooner, and Later. By the end of every day, do this "Four-D" process on everything in your inbox: If you can "do" it, do it on the spot. If you can "delegate" it, make it an assignment for someone else and CC yourself, then file the message in a future day's folder for follow-up. If you can or must wait on something, "defer" it by filing it in the appropriate day's folder and worry about it then. If it can't be done, delegated, or deferred, then just "delete" it. In my case, I delete to an archive for future reference. At the start of each day, process all the items in the "sooner" folder and all the items in that day's box. On Friday, review everything in the "later" folder. Doing this every day, you never lose email, you never miss something important. You never miss anything.

Second, ignore your telephone and practice the concept of "one conversation at a time," which is a lesson of the "Fierce Conversations" school of thought [3]. If you are talking to a live person who took the time to walk to your desk, give that person your attention and don't even look to see who is calling on the telephone. If a second person comes to talk to you while you're talking to the first, don't put the first person in sleep status to process the new interrupt first. Finish the first conversation and move to the second. Check voice messages several times per day and return calls, and in general treat the telephone like it's just a voice-activated email system. Don't work for your telephone, make it work for you.

Third, keep a running tab on everything you have to do, that you've asked others to do, and that you want to track. Carry a notebook and write things down. If you don't have a notebook, write things down on your hand (I'm well-known for my "palm pilot" that has a tendency to reboot when I wash my hands). Don't trust your brain. Move everything from your notebook (or hand) to your wiki. The wiki should be something only you see, and should lay out the same way as your email inbox folders, but with more range. Days of the week, sooner and later, but also months and years out. Keep track of ideas you have that might be worth exploring next year. Take special note of anything you have to do in the morning to prevent getting fired. Consult and update this wiki when you start your day and when you end it, so you know what you've done and what you have to do next, while never losing sight of what your long-term issues are.

Finally, understand your own priorities. My priorities are, in order: people on my team, my customers, my managers, human resources and finance, other people in my organization, and external entities like vendors. My golden rule: if one of my own people needs me, they are my priority. Their job is sysadmin. My job is taking care of them.

Using the System

I've been using this system for three years with great success, and I'm pleased to share it with you. My goal is to understand the things that are really important and to be able to absorb and process all the relevant information flows in my organization. In the modern digital age, information is both faster and has more volume than the average person can handle. A reliable system is like a fulcrum, it helps me to lift more than I'm actually capable of doing. This system helps me to understand what's important and to focus where my efforts matter most, which is usually in support of my team. I'm the manager, and this is how I do my job.

[1] T. Limoncelli, *Time Management for System Administrators,* O'Reilly Media, 2005.

[2] D. Allen, *Getting Things Done,* Penguin, 2002.

[3] S. Scott, *Fierce Conversations: Achieving Success at Work and in Life One Conversation at a Time,* The Berkley Publishing Group, 2002.

Workshops and Publications

PETER H. SALUS

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

Although I mentioned the first graphics workshop a few months ago, after 1985 both the number of workshops and the number of publications increased dramatically over the next decade. And not all of the publications were on paper. Here's the tale.

UNIX NEWS may have been the first UNIX publication outside of AT&T, but only by a bit. On April 30, 1976, it was announced that Lew Law of the Harvard Science Center would "undertake the task of duplicating and distributing the manuals for UNIX." That was "Sixth edition," or v6. It was the beginning of external publication.

The same issue of *UNIX NEWS* carried an article by Bill Mayhew (of the Children's Museum in Boston) on "How to fix your PDP-11/40's static electricity problems for 49 cents (plus tax)." And the next issue (May-June 1976) announced "the first mailing from the software exchange." Software exchange?

Lew Law supplied software from Harvard, and Mike O'Brien did the duplication and mailing of tapes. Freely redistributed software in 1976! And there was a second distribution in November 1976, containing software from the RAND Corporation, the Naval Postgraduate School, UCSD, Yale, and UIUC. There was a third distribution in May 1977, and contributed software was assembled and distributed on tape until 1989.

Conferences and Workshops

For the decade following the June 1975 meeting in New York, there were two USENIX conferences each year, one in the east (New York, Cambridge, Chicago, Urbana, Newark (DE), Toronto, Austin) and one in the west (Monterey, Berkeley, Menlo Park, Santa Monica, Boulder, San Francisco). Some years there were three.

The first separately published *Proceedings* was for Toronto (July 13–15, 1983), and the second was for Salt Lake City (June 13–15, 1984). There were also proceedings for the "Unicom" conferences—USENIX and /usr/group co-located—San Diego, January 1983, and Washington, DC, January 1984. Proceedings appeared for nearly 20 years. I miss them, although I realize that bits have superseded paper.

In 1984 the (newly elected) USENIX Board announced three "limited enrollment" workshops: Distributed Systems, Communications and Networking, and Graphics. For organizational reasons, the Communications and Networking Workshop was cancelled. Distributed Systems was held in what proved to be an unsatisfactory venue in Newport, RI, although nearly all of the 100 attendees regarded it as "clearly worthwhile" and "should be repeated." The "UNIX and Computer Graphics Workshop," held in Monterey, CA, was a great success.

The report on "Distributed UNIX" by Veigh S. Meer (a transparent pseudonym) appeared in *;login:* 9.5 (November 1984), pp. 5–9.

A Digression on ;login: *and on Manuals*

The May–June 1977 issue of *UNIX NEWS* was its last. As of July 1977, the publication was *;login:*. Mel Ferentz had been phoned by an AT&T lawyer and told that the group (it still had no name) could not use "UNIX" without permission from Western Electric. At a meeting at Columbia's College of Physicians and Surgeons (May 24–27, 1978) a committee was set up to propose bylaws for an organization. Margaret Law, then at Harvard and Radcliffe, coined the name USENIX.

UNIX NEWS was succeeded by *;login:*. As Dennis Ritchie explained, "The **;** was utilitarian. During most of the early '70s the most popular terminal was the Teletype model 37. The sequence **<esc>;** put it in full-duplex mode so the terminal didn't print characters locally, but let the system echo them. So this sequence was put into the greeting message."

Through the 1970s, AT&T UNIX came with next to no documentation; hence Lew Law's offer of 1976. By the time Berkeley UNIX (BSD) was developed, diverging from AT&T UNIX, manuals were in real demand. The Computer Systems Research Group (CSRG) had no way of coming to grips with the demand, so the USENIX office, now in El Cerrito, just north of Berkeley, took on the printing and distribution. Thus, in April 1984, *;login:* featured an announcement of the availability of the 4.2BSD manuals in five volumes. They sold out quickly. In February 1985, a new printing was announced. A third and a fourth printing ensued in late 1985 and early 1986. In late 1986, 4.3BSD followed with an index volume (thanks to Mark Seiden) added. (The 4.4BSD set was published by O'Reilly.)

As these were CSRG documents, printed and sold by USENIX, I've never been certain whether to consider them USENIX publications.

Back to Workshops

Six papers from the 1984 Graphics Workshop appeared in *;login:* 10.4, October-November 1985 (pp. 22–83), along with a CFP for the Second Workshop, to be held in December in Monterey. Embarrassingly, there were only four issues of *;login:* in 1985. One of the consequences of this was the replacement of the Executive Director (who served for less than a year) by the present writer.

One of the things the Board asked of me in the spring of 1986 was an increase in the number of workshops and of publications. Among the items on my desk was a manila envelope containing the papers from the 1985 Graphics Workshop.

I consulted with Tom Strong and he had sheets with headers and footers printed. I hired Steven Katz to paste up the articles, and we sent the bundle off to be printed: the Association's first workshop proceedings appeared in late summer 1986.

With that, and the third Graphics Workshop under way, Rob Kolstad suggested a Large Installation Systems Administrators' Workshop, and Kirk McKusick and John S. Quarterman suggested a POSIX Workshop as well as one on C++ and a fourth Graphics Workshop for 1987.

Just over 50 people attended the first LISA in Philadelphia (April 9–10); about 30 were admitted to the POSIX event in Berkeley (October 22–23), where several thousand comments and corrections were appended to the P1003.1 draft. The Fourth Graphics Workshop was held in Cambridge, MA, October 8–9, and C++ was held in Santa Fe, November 9–10, rounding off a busy 1987.

Over the past decades, there have been a number of major changes where "gatherings" are concerned: first, the USENIX Association dropped down to a single annual meeting; parallel to that, the number of small- or medium-sized workshops has blossomed. I personally think this is less than wonderful. At a large semiannual meeting in the late 1980s or the 1990s, one might wander into a session on a new OS or a bizarre language or on networking hundreds of small CPUs. You might not have had colorful acronyms, like SOUPS or WOOT or CSET or JETS or HOTSEC, but you had a very large number of interesting people in one place.

And you never knew whom you might meet in a corridor or at the Scotch BoF.

The last big change was moving from print on paper to bits.

R.I.P. COMPUTING SYSTEMS

One of the things the USENIX Board wanted in 1986-87 was a journal that concerned software more than hardware. Think of *CACM* and that "M" for Machinery. So I spoke to folks at several academic publishers and came in with a proposal for a quarterly journal. It was announced in *;login:* 12.6 (November-December 1987). It first appeared (Mike O'Dell, Editor in Chief) the next year, published by University of California Press.

I was Managing Editor for its whole nine-year lifespan. Mike was superseded by Dave Presotto after a brilliant seven years. MIT Press took over as publisher. *Computing Systems:* I could wax nostalgic and itemize authors and articles, but I'll refrain from doing so.

However, let me note that in 1988, *CS* published an article by Mike Lesk, "Can UNIX Survive Secret Source Code?" In 1990 an entire issue (accompanied by a CD) was devoted to music. In 1992 there was an entire issue on Internet search mechanisms. And in 1996, a final issue on distributed objects.

I wish it were still being published.

Everything changes: the things we like and those we don't.

Interview with Dr. Dan Geer

RICHARD THIEME

Richard Thieme (www.thiemeworks.com, neuralcowboy@gmail.com) is an author and professional speaker focused on the deeper implications of technology, religion, and science for twenty-first-century life. He has published hundreds of articles, dozens of short stories, three books, and has delivered hundreds of speeches. A novel, *FOAM,* is now available, and "A Richard Thieme Reader," collecting selected fiction and non-fiction, will be published soon. rthieme@thiemeworks.com

Dr. Dan Geer was at the time of this interview [Fall 2000] the Chief Technical Officer of @stake, a digital security consulting firm, and had recently been elected President of the USENIX Association. USENIX is a 10,000-member organization comprising engineers, system administrators, scientists, and technicians working on the cutting edge of the computing world. Geer, who holds a ScD from Harvard University, was a professor at the Harvard School of Public Health and participated in MIT's Project Athena and the development of the X Window System and Kerberos. He held executive positions at Open Market, Inc., OpenVision Technologies (now Veritas), and CertCo, the leading online risk assurance authority. Geer has testified before the House Science Committee and Subcommittee on Technology regarding public policy in the age of electronic commerce. He is currently (2015) the CISO of In-Q-Tel, a research and development arm of the CIA.

RT: Dan, you were just chosen President of USENIX. What's the significance of that for you? What's your vision for USENIX?

DG: I think the best way to thank somebody is to help them out. I got a lot out of that place, and I am trying to put something back. That may sound corny, but it's a fact. I guess my momma raised me right.

In lots of ways, USENIX made me what I am. USENIX has kept me from getting too satisfied. People who get satisfied stop growing. People who are never satisfied are always curious. They keep growing.

When I try to hire new people, I put a checkmark on the page when I realize that the person I'm interviewing is never satisfied with what they know or can do. The smartest people feel as if they know the least. Over and over again, USENIX told me things I didn't know I didn't know.

I highly recommend that any young person starting out, or even someone not so young, should work with program committees for conferences, editorial boards for journals, anything where the interesting traffic is concentrated in your direction. It's almost impossible to lose if you're serious about putting in the effort. Otherwise you have to search for the best work and it's rarely in one place or conveniently indexed. It's much more difficult to learn to swim if you're not in the water.

That's what I've gotten out of it. What I am trying to put back in—maybe it's my heritage, that I'm a security guy—but I'm a professional paranoid. If you think that good times are permanent, you guarantee that they won't be. USENIX, like everyone else, must be aware of what's changing, what old opportunities are being eclipsed and what new ones are showing up. As President I intend to push us pretty hard to obsolete our products before someone else does, just as Andy Grove and Jack Welch try to do.

Even for a nonprofit in very good shape like USENIX, it's essential to obsolete our product or someone else will. We need to bring on new conferences. The established conferences in our game more than pay for themselves, while the brand new ones don't even come close. So there is a cross-subsidy: what you already do well allows you to take risks in things you don't do so well. I am pushing pretty hard in that direction.

In the venture capital arena, investors want to invest in companies that go straight down or straight up. They don't want a 2% grower that makes it impossible to get your money out yet you can't write it off. In some sense, intellectual capital has the same characteristics—I want prompt failure or prompt success. I don't want to spend ten years on something that finally struggles to its feet. As a wise person said, the cost of anything is a foregone alternative. That's the kind of paranoia I am trying to bring to the job.

I have always tried to pick jobs where my colleagues would challenge me. The best jobs I have had, I knew I would be embarrassed from day one.

RT: It's critical to keep moving out of your comfort zone, to keep yourself on the edge.

DG: Yes. I am not an adrenaline sports guy, but maybe it's the same urge applied in a way that has greater long-term value.

[Editor's note: There was a lot more in this interview, which will someday appear in a collection of Richard Thieme's interviews. We include an exchange near the end of the interview, as we found it quite prescient.]

RT:... and anomaly detection and misuse detection. So maybe in some gray area we must compromise, and that's where risk management comes in. We may never achieve a stasis at the level of totalitarian control, but we are moving in that direction.

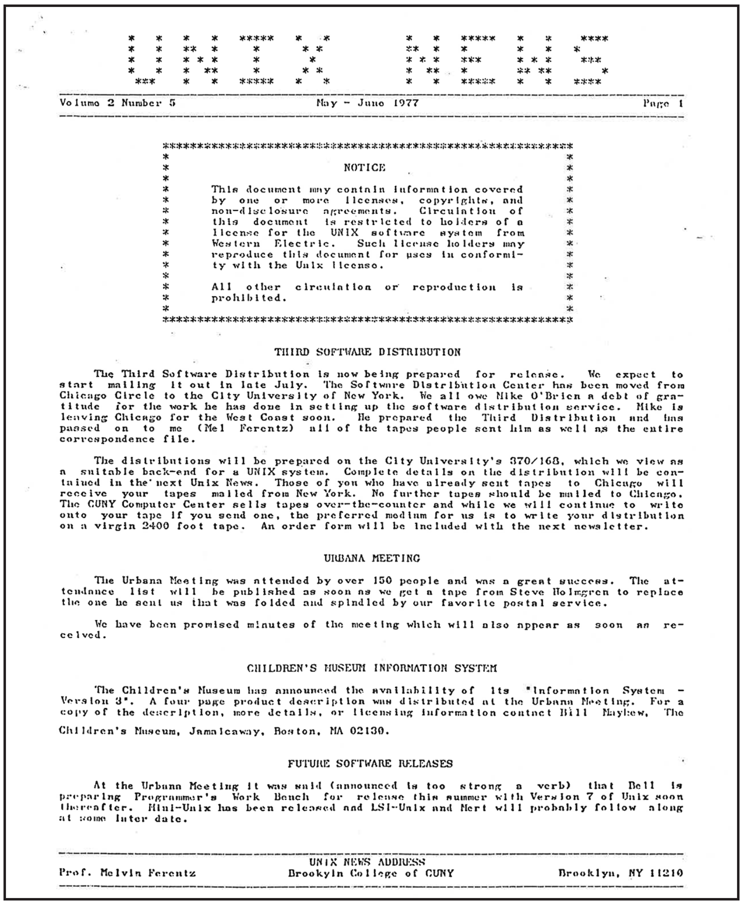
DG: Yes. It is unlikely that someone will come to you personally and take your privacy away, but children do not have an expectation of privacy. They only develop it later. So if you don't know that you never had it, how much of a fight will you put up when you don't get it?

I don't think it's possible to go much further in our technological world on a "small is beautiful"/egalitarian basis. To continue to rail that way is to give away the lead time we have to modify the coming culture rather than allow it to wash over us like a wave.

Source: Richard Thieme, "All Geered Up," *Information Security*, October 2000, vol. 3, no. 10, pp. 86–92.

UNIX News

Volume 2, Number 10, May–June 1977



```
*   *   *   *   *****   *   *        *   *   *****   *   *    ****
*   *   **  *     *      * *         **  *   *       *   *   *
*   *   * * *     *       *          * * *   ***     * * *    ***
*   *   *  **     *      * *         *  **   *       ** **       *
 ***    *   *   *****   *   *        *   *   *****   *   *   ****
```

Volumo 2 Number 5 May - June 1977 Page 1

```
***********************************************************
*                                                         *
*                         NOTICE                          *
*                                                         *
*      This document may contain information covered     *
*      by one or more licenses, copyrights, and           *
*      non-disclosure agreements. Circulation of          *
*      this document is restricted to holders of a        *
*      license for the UNIX software system from          *
*      Western Electric. Such license holders may         *
*      reproduce this document for uses in conformi-      *
*      ty with the Unix licenso.                          *
*                                                         *
*      All other circulation or reproduction is           *
*      prohibited.                                        *
*                                                         *
***********************************************************
```

THIRD SOFTWARE DISTRIBUTION

The Third Software Distribution is now being prepared for release. We expect to start mailing it out in late July. The Software Distribution Center has been moved from Chicago Circle to the City University of New York. We all owe Mike O'Brien a debt of gratitude for the work he has done in setting up the software distribution service. Mike is leaving Chicago for the West Coast soon. He prepared the Third Distribution and has passed on to me (Mel Ferentz) all of the tapes people sent him as well as the entire correspondence file.

The distributions will be prepared on the City University's 370/168, which we view as a suitable back-end for a UNIX system. Complete details on the distribution will be continued in the next Unix News. Those of you who have already sent tapes to Chicago will receive your tapes mailed from New York. No further tapes should be mailed to Chicago. The CUNY Computer Center sells tapes over-the-counter and while we will continue to write onto your tape if you send one, the preferred medium for us is to write your distribution on a virgin 2400 foot tape. An order form will be included with the next newsletter.

URBANA MEETING

The Urbana Meeting was attended by over 150 people and was a great success. The attendance list will be published as soon as we get a tape from Steve Holmgren to replace the one he sent us that was folded and spindled by our favorite postal service.

We have been promised minutes of the meeting which will also appear as soon as received.

CHILDREN'S MUSEUM INFORMATION SYSTEM

The Children's Museum has announced the availability of its "Information System - Version 3". A four page product description was distributed at the Urbana Meeting. For a copy of the description, more details, or licensing information contact Bill Mayhew, The Children's Museum, Jamaicaway, Boston, MA 02130.

FUTURE SOFTWARE RELEASES

At the Urbana Meeting it was said (announced is too strong a verb) that Bell is preparing Programmer's Work Bench for release this summer with Version 7 of Unix soon thereafter. Mini-Unix has been released and LSI-Unix and Mert will probably follow along at some later date.

UNIX NEWS ADDRESS
Prof. Melvin Ferentz Brookyln College of CUNY Brooklyn, NY 11210

UNIX News, volume 2, number 5, published in June 1977 by Professor Melvin Ferentz of Brooklyn College of CUNY, was the last issue of the newsletter under that title. In July 1977, the first issue of *;login: The UNIX Newsletter,* appeared. These excerpts from *UNIX News* have been reproduced as they appeared in the original, including any typographic errors. *Note: We have not included the mailing list and other addresses and telephone numbers that appeared in the original issue.*

Third Software Distribution

The Third Software Distribution is now being prepared for release. We expect to start mailing it out in late July. The Software Distribution Center has been moved from Chicago Circle to the City University of New York. We all owe Mike O'Brien a debt of gratitude for the work he as done in setting up the software distribution service. Mike is leaving Chicago for the West Coast soon. He prepared the Third Distribution and has passed on to me (Mel Ferentz) all of the tapes people sent him as well as the entire correspondence file.

The distributions will be prepared on the City University's 370/168, which we view as a suitable back-end for a UNIX system. Complete details on the distribution will be continued in the next Unix News. Those of you who have already sent tapes to Chicago will receive your tapes mailed from New York. No further tapes should be mailed to Chicago. The CUNY Computer Center sells tapes over-the-counter and while we will continue to write onto your tape if you send one, the preferred medium for us is to write your distribution on a virgin 2400 foot tape. An order form will be included in the next newsletter.

Urbana Meeting

The Urbana Meeting was attended by over 150 people and was a great success. The attendance list will be published as soon as we get a tape from Steve Holmgren to replace the one he send us that was folded and spindled by our favorite postal service.

We have been promised minutes of the meeting which will also appear as soon as received.

Children's Museum Information System

The Children's Museum has announced the availability of its "Information System—Version 3." A four page product description was distributed at the Urbana Meeting. For a copy of the description, more details, or licensing information, contact Bill Mayhew.

Future Software Releases

At the Urbana Meeting it was said (announced is too strong a verb) that Bell is preparing Programmer's Work Bench for release this summer with Version 7 of Unix soon thereafter. Mini-Unix has been released and LSI-Unix and Mert will probably follow along at some later date.

Vrije Universiteit, Amsterdam

From E.G. Keizer

We are using UNIX on our PDP 11/45 for almost a year now and are very enthusiastic about it. Our system is somewhat overloaded but we hope that the disk drives we ordered will help to solve the problem.

Lately we found a "bug" in the UNIX kernel. One of our users was having troubles with his program that was switching back and forth between single and double precision Floating Point node. We discovered that the F.P. registers are saved in the node the F.P. processor has at the moment the program is stopped. This means that the low order 32 bits of the users double precision registers were not saved whenever his program was stopped in single mode. By adding setd instructions in m45.s just before the lines where the F.P. registers are moved to and from _u, we solved the problem. Consequently the F.P. registers are always stored in double mode. The programs db and cdb will have to be changed to reflect the new situation.

A few months ago somebody noticed that the times stated by the time command were somewhat off. Time expects that the system command times returns process and system times in 60ths of seconds. But since we have a 50 Hz power supply, times returned those times in 50ths of seconds. He changed time.s according to our situation.

We had some problems with the pipe mechanism. When several processes were writing simultaneously on one pipe their messages got intermixed if the pipe pointers reached the end of the pipe buffer.

In case somebody is interested in a driver for the old DEC DM11 multiplexer, we would be glad to send a copy of our driver.

Katholieke Universiteit Nijmegen

From George Rolf

If no one else wrote you about the matter before, here is our fix to the ttyn(III) problem mentioned in the February issue of UNIX News. I found the bug about 3 months ago.

```
After the line at reads:  mov   buf+2,(sp)
I inserted                mov   buf,r1
                          sys   stat;dev;buf
                          bes   er1
                          cmp   buf,r1
                          bne   cr1
```

A similar change has to be made to nroff(I), file: s7/nroff1.s. This file contains a slightly different version of ttyn. The following commands may be considered candidates for recompilation: em, goto, pr, rn, login, mail, mesg, ps, who.

Problems with creat system call on Unix version 6

From George Goble, Purdue University

We have discovered two problems with the "creat" system call. The following sequence of commands will cause "orphaned" files (files that are not in any directory) to be created:

```
chdir /tmp
mkdir a
chdir a
rmdir /tmp/a
ls -l / >orphan
chdir /
```

The rmdir causes the link count for the /tmp/a inode to goto zero, however the inode is not deallocated because it is the shell's current directory. As this point one can create files in the current directory. One (except super user) cannot create directories in the current directory because mkdir does a stat on ".." which does not exist. Upon doing a chdir /, the reference count for the old current directory goes to 0, causing deallocation of its inode and stranding the newly created files.

The second problem occurs when the maknode call in creat() fails due to no inodes on the device. Namei leaves the last directory inode in the pathname locked because a return is executed after the maknode failure. The next process to reference the locked inode will go to sleep (and hang!) with PINOD (-90) priority.

The fix for the first problem consists of adding an error return if the current directory inode has a link count of zero. Below is a copy of the existing creat() in /usr/sys/ken/sys2.c and the revised one.

Existing creat() in /usr/sys/ken/sys2.c

```
creat()
{
    register *ip;
    extern uchar;

    ip = namei (&uchar, 1) ;
    if(ip == NULL) {
           if(u.u_error)
                   return;
           ip = maknode(u.u_arg[1]&07777&(~ISVTX) ) ;
           if = (ip==NULL)
                   return;
           open1(ip, FWRITE, 2);
    ] else
           open1(ip, FWRITE, 1);
}
```

Modified create() in /usr/sys/ken/sys2.c

```
create()
{
    register *ip;
    extern uchar;

    ip = namei (&uchar, 1) ;
    if(ip == NULL) {
            if(u.u_error)
                    return;
            if((u.u_cdir->i_nlink == 0) && (fubyte(u.u_arg[0])!= '/')) {
                    u.u_error = ENOENT;
err:                iput(u.u_pdir); /* namei left parent dir locked */
                    return;
            }
            ip = maknode(u.u_arg[1]&07777&(~ISVTX) ) ;
            if = (ip==NULL)
                    goto err;
            open1(ip, FWRITE, 2);
    } else
            open1(ip, FWRITE, 1);
}
```

University of Glasgow

From Alistair C. Kilgour

I am writing to let you know of the formation of a U.K. Unix Users Group. The first meeting took the form of a Colloquium at Glasgow University on Friday 27th May, attended by about 40 people. Short Talks were presented on aspects of the kernel including the scheduler and the buffer cache system, the structure of CAC "Network Unix", the features of the Carnegie Mellon INGRES relational database system, and some early experience with the Toronto graphics software. During the afternoon session the User Group was formally constituted. Two officials were elected, myself as chairman and Peter Gray of Aberdeen University as Secretary and Newsletter Editor. It was not felt necessary at the present time to elect any form of executive committee.

It was agreed that an attempt should be made to constitute the group as a Special Interest Group under the umbrella of DECUS U.K. We are seeking approval of this move both from Bell and from the DECUS Executive Board. DECUS have agreed to handle distribution of the U.K. Unix Newsletter, and will undertake to send it only to accredited Unix license-holders, so we don't foresee any problems with Bell. General information about meetings etc., will be published in the DECUS U.K. Newsletter, but all system-specific material will be restricted to the SIG publication.

On the question of languages the appearance in the U.K. of the Princeton RT11 FORTRAN implementation was generally welcomed, at least by the "engineering" interests. The availability (subject, of course, to having purchased appropriate DEC licenses) of a good FORTRAN which can be configured for the full range of hardware is bound to enhance the appeal of Unix in non-computer-science departments.

Software Standards

Concern was expressed on several points in the area of system standards, particularly in distributed software. Among the points raised were the following:

(i) User group standard software: since it is increasingly difficult for U.K. users to attend personally any of the U.S. meetings, it would be nice if the views of users outside the U.S. could be sought before a piece of software or a system mod. is adopted as a standard. In the case of the Yale Shell, we are all delighted with it, but future proposals could be more controversial.

(ii) Assumed hardware: wherever possible distributed software should be configured for a "standard" system, with instructions for modifications required for other hardware. Assumed conventions about pathnames, etc., should be made explicit.

(iii) Documentation: manual pages should be in 'nroff' form, using the standard 'tmac.naa' macro definitions, and have extension '1' or '6'. Other documentation should include any required nroff macro definitions.

(iv) System calls: the adoption of the 'terms' system call as a standard was suggested. The group from 56 to 63 should be reserved for locally added system calls, and no distributed software should make any assumptions about the system calls in this range.

Software Distribution

The meeting agreed that Glasgow University Computing Science Department should enter negotiations with a view to becoming a software distribution centre for the U.K. We have three exchangeable HK05 drives, and by the end of July should have an 800/1600 bpi magnetic tape drive. We will also act as a collection centre for software which U.K. users with to contribute to the U.S. distribution centre.

If any U.S. Unix addicts are visiting the U.K. this summer, please drop in and see us. (I'm sure that goes for all of the U.K. Unix sites).

Seeing Stars

DAVID BEAZLEY

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

As you read this, Python 3.5 should be hitting the streets with a wide assortment of new features and even some new syntax. "New syntax?" you ask. Why yes. Even though Python has been around for more than 25 years now, it continues to evolve and sprout surprising new features from time to time. In this month's installment, I'm going to look at a seemingly minor part of Python that turns out to be fairly useful—the use of `*` and `**` in function arguments, function argument passing, and data handling.

You Want an Argument?

Traditionally, `*` and `**` have been used to write functions that accept any number of positional or keyword arguments. For example, this function accepts any number of positional arguments, which are passed as a tuple to `args`:

```
>>> def f(*args):
...     print(args)
...
>>> f(1,2,3)
(1, 2, 3)
>>> f(1)
(1,)
>>> f(4,5)
(4, 5)
>>>
```

This function accepts any number of keyword arguments, which are passed to `kwargs` as a dictionary:

```
>>> def g(**kwargs):
...     print(kwargs)
...
>>> g(color='red', size='huge')
{'color': 'red', 'size': 'huge'}
>>> g(xmin=0, xmax=-10, title='Plot')
{'xmin': 0, 'xmax': -10, 'title': 'Plot'}
>>>
```

The `*args` and `**kwargs` can be combined with other arguments and even used together as long as they go at the end of the argument list and the keyword arguments appear last. For example:

```
def h(x, y, *args, **kwargs):
    ...
```

A common use of `*args` and `**kwargs` is in writing code that's meant to be very general purpose. For example, consider this class definition that makes it easy for someone to define simple data structures:

```
class Structure(object):
    _fields = ()
    def __init__(self, *args):
        if len(args) != len(self._fields):
            raise TypeError('Expected %d arguments' % len(self._
fields))
        for name, val in zip(self._fields, args):
            setattr(self, name, val)

# Examples
class Date(Structure):
    _fields = ('year', 'month', 'day')

class Address(Structure):
    _fields = ('hostname', 'port')
```

Sometimes **kwargs is used to write functions that take a large number of options that you want specified by keyword only. For example:

```
def config(**options):
    outfile = options['outfile']     # Required argument
    level = options.get('level', 0)  # Optional argument
    ...

config(outfile='output.txt', level=20)    # Ok
config('output.txt', 20)                  # Error.
```

Passing Argument

The * and ** syntax are also used to pass data as arguments to functions. For example, suppose you have this function:

```
def f(x, y, z):
    ...
```

If you already have a sequence of arguments or a dictionary of keywords, you can pass them as follows:

```
a = (1, 2, 3)
b = { 'x': 1, 'y': 2, 'z': 3}

f(*a)      # Same as f(1, 2, 3)
f(**b)     # Same as f(x=1, y=2, y=3)
```

Both of these conventions can be especially useful when working with data that you have already obtained somehow but that you want transformed into another form. For example, suppose you have a list of tuples and a class definition like this:

```
stocks = [
   ('IBM', 50, 91.25),
   ('HPQ', 75, 37.23),
   ('MSFT', 100, 47.80)
]
```

```
class Stock(object):
    def __init__(self, name, shares, price):
        self.name = name
        self.shares = shares
        self.price = price
```

You can easily convert the list into instances using a statement like this:

```
stocks = [Stock(*s) for s in stocks]
```

The use of * also enables some unusual tricks. For example, consider this example of "unzipping" data:

```
>>> a = ['name', 'shares', 'price']
>>> b = ['IBM', 50, 91.25]
>>> # Zip the two sequences into a list of tuples
>>> c = list(zip(a,b))
>>> c = [('name', 'IBM'), ('shares',50), ('price', 91.25)]
>>> # Unzip a list of tuples into separate sequences
>>> d, e = zip(*c)
>>> d
('name', 'shares', 'price')
>>> e
('IBM', 50, 91.25)
>>>
```

Needless to say, that last step with zip(*c) might require a bit more study (left as an exercise).

Keyword-Only Arguments

Python 3 introduced an extension to the * syntax that makes it easier to have keyword-only arguments. Specifically, named arguments are allowed to appear after an argument with *. For example:

```
def receive(maxsize, *, block=True):
    ...

msg = receive(1024)                   # OK
msg = receive(1024, block=False)      # OK
msg = receive(1024, False)           # Error

def total(*items, initial=0):
    total = initial
    for it in items:
        total += it
    return total

a = total(1,2,3, initial=100)      # a <- 106
```

This ability to have named keyword-only arguments can be a useful way to clean up library code that might otherwise depend on `**kwargs`. For example, the `config()` function from earlier could be rewritten as follows:

```
def config(*, outfile, level=0):
    ...
```

This version will produce better error messages, have a more useful help screen, and involve much less code related to handling the arguments. Keyword-only arguments are good.

Wildcard Unpacking

If you have a tuple, it is easy to unpack into separate variables. For example:

```
address = ('www.python.org', 80)

hostname, port = address        # Unpack
```

This all works well as long as the number of items in the tuple exactly matches the number of variables specified—if not, you get an error. Python 3 allows you to use the * as a wildcard in unpacking. For example:

```
>>> row = ('Elwood', 'Blues', '1060 W Addison', 'Chicago', 'IL',
'60613')
>>> first, last, *rest = row
>>> first
'Elwood'
>>> last
'Blues'
>>> rest
['1060 W Addison', 'Chicago', 'IL', '60613']
>>> first, last, *rest, zipcode = row
>>> first
'Elwood'
>>> last
'Blues'
>>> zipcode
'60613'
>>> rest
['1060 W Addison', 'Chicago', 'IL']
>>>
```

Notice how all of the extra values are simply placed in a list. Wildcard unpacking can be particularly useful if you're working with rows of data of varying length but are only interested in some of the values. For example:

```
rows = [
    (1, 2),
    (3, 4),
    (5, 6, 'x'),
    (7, 8, 'x', 'y'),
    (9, 10)
]

for x, y, *extra in rows:
    ...
```

Unpacking and Argument-Passing Extensions

Python 3.5 extends the capabilities of * and ** in some new and interesting directions. First, you can use both operations more than once when making function calls. For example:

```
def f(a, b, c, d):
    ...

x = (1, 2)
y = (3, 4)
f(*x, *y)       #  Same as f(1, 2, 3, 4)

x = { 'a': 1, 'b': 2}
y = { 'c': 3, 'd': 4}
f(**x, **y)     #  Same as f(a=1, b=2, c=3, d=4)
```

These extensions simplify code that previously had to assemble the arguments by hand. For example, in previous versions of Python, you would have had to write the following:

```
f(*(x+y))
f(*(tuple(x)+tuple(y)))    # Safer version to make sure types
                           # match in +

kwargs = dict(x)           # Make a copy of x
kwargs.update(y)           # Merge in values from y
f(**kwargs)
```

You can also perform unpacking when creating list, tuple, set, and dictionary literals. For example:

```
a = [1, 2]
b = [ *a, 3, 4]        # b = [1, 2, 3, 4]
c = [3, *a, 4]         # c = [3, 1, 2, 4]
d = [3, *a, *a, 4]     # d = [3, 1, 2, 1, 2, 4]

m = { 'x': 1, 'y': 2 }
n = { **m, 'z': 3 }    # n = {'x':1, 'y':2, 'z':3 }
```

In such unpacking, later elements will silently replace earlier elements if there happen to be any duplicates. For example:

```
a = { 'x': 1, 'y': 2 }
b = { 'x': 3, 'z': 4 }
c = { **a, **b }        # c = { 'x':3, 'y':2, 'z':4 }
```

Although these enhancements look minor, they do enable certain kinds of new operations. It is now easy to merge dictionaries as a single expression as shown above. This can extend naturally into operations involving lists of dictionaries and other structures. For example:

```
s1 = [
    {'x': 1, 'y': 2},
    {'x': 3, 'y': 4},
    {'x': 5, 'y': 6}
]

s2 = [
    {'z': 10, 'w': 11 },
    {'z': 12, 'w': 13 },
    {'z': 14, 'w': 15 }
]

merged = [ { **i1, **i2 } for i1, i2 in zip(s1, s2) ]
# merged = [
#     { 'x': 1, 'y': 2, 'z': 10, 'w': 11},
#     { 'x': 3, 'y': 4, 'z': 12, 'w': 13},
#     { 'x': 5, 'y': 6, 'z': 14, 'w': 15}
# ]
```

This change also enables a common dictionary type transformation that I find myself performing with some regularity. For example, suppose you have some raw dictionary data read from a file such as this:

```
rows = [
 {'name': 'AA', 'price': '32.20', 'shares': '100'},
 {'name': 'IBM', 'price': '91.10', 'shares': '50'},
 {'name': 'CAT', 'price': '83.44', 'shares': '150'},
 {'name': 'MSFT', 'price': '51.23', 'shares': '200'},
 {'name': 'GE', 'price': '40.37', 'shares': '95'},
 {'name': 'MSFT', 'price': '65.10', 'shares': '50'},
 {'name': 'IBM', 'price': '70.44', 'shares': '100'}
}
```

Now suppose you wanted to apply a conversion to some of the values (e.g., convert shares to an integer and price to a float). You can do this:

```
conversions = [ ('shares', int), ('price', float) ]
converted = [ {**row, **{name:func(row[name]) for name, func
in conversions}}
           for row in rows ]
```

This does exactly what you want, although I'm willing to concede that it might be too clever for its own good. The alternative is to unwind it to this:

```
converted = []
for row in rows:
    newrow = dict(row)
    for name, func in conversions:
        newrow[name] = func(row[name])
    converted.append(newrow)
```

Needless to say, that's not nearly as clever nor preserving of one's future job security.

More Information

If you're intrigued by some of the new uses of `*` and `**kwargs`, more information can be found in various PEPs. For example, PEP 448 describes the generalized unpacking features added to Python 3.5 [1]; PEP 3102 describes keyword-only arguments [2]; and PEP 3132 describes the wildcard unpacking of sequences [3].

These are not the only syntax changes to Python 3.5. In future installments, we'll look at some of the new features added to the language. In the meantime, you might take a look at the "What's New in Python 3.5" document [4].

References

[1] PEP 448: https://www.python.org/dev/peps/pep-0448/.

[2] PEP 3102: https://www.python.org/dev/peps/pep-3102/.

[3] PEP 3132: https://www.python.org/dev/peps/pep-3132/.

[4] https://docs.python.org/dev/whatsnew/3.5.html.

Practical Perl Tools

Blog, Can We Talk?

DAVID N. BLANK-EDELMAN

David Blank-Edelman is the Technical Evangelist at Apcera (the comments/views here are David's alone and do not represent Apcera/Ericsson). He has spent close to 30 years in the system administration/DevOps/SRE field in large multiplatform environments, including Brandeis University, Cambridge Technology Group, MIT Media Laboratory, and Northeastern University. He is the author of the O'Reilly Otter book *Automating System Administration with Perl* and is a frequent invited speaker/organizer for conferences in the field. David is honored to serve on the USENIX Board of Directors. He prefers to pronounce Evangelist with a hard "g."
dnblankedelman@gmail.com

According to some figures (those at http://w3techs.com/technologies/details/cm-wordpress/all/all to be precise), WordPress powers 24.2% of the sites on the Internet. I don't have any reason to doubt that number. WordPress (WP) has lots of great things going for it when you are looking to bring up a Web site containing dynamic content. Beginners can grasp it fairly quickly, it has a huge and vibrant ecosystem, a strong development effort, oh, and it's free. It's my "goto" tool when someone comes to me and needs my help building a Web site for their aardvark repair company or whatnot. It may have started as blogging software, but it has evolved far beyond that over the years into a reasonable Web development platform.

So why am I giving you a sales pitch for WP in a Perl column? In addition to making my bias clear, I figure if it powers close to a third of the Web sites on the planet, it could be a good idea to learn to interact with it via Perl. Why would you want to do this? For me, the best reasons center around being able to easily extract information posted on a WordPress site or, even better, the ability to post external sources of information right to a WordPress site. For example, let's say you had a process for generating sales reports that took hours of heavy-duty computation on a massive data warehouse. It might be very handy to post the results to an internal WordPress site every day for people to be able to easily access.

The good news is we are going to be able to draw strongly on past columns and knowledge for this effort. One quick prerequisite: I'm going to make the assumption that you have at least a passing familiarity with WordPress (you know it has posts, pages, and users, and you know how to install plugins) and administrative access to a working up-to-date WP site.

Here's What We Are Not Going to Do

There are lots of inelegant ways we could interact with WP (some of which we've explored in this column). For example, we could use something like WWW::Mechanize or Selenium to pretend to be Web browsers to screenscrape the pants off the site or fake like we are typing/clicking. I could make you more nauseous by noting that WordPress has a MySQL backend (plus access to a file system) so we could just whip out DBI and go to town. Nope, not going to do it.

A much more reasonable approach might be to use the closest thing WordPress has had to an external API: the XML-RPC interface it provides via the xmlrpc.php file. And, indeed, there have been modules written in days of yore like WordPress::XMLRPC that use this API. Even though XML-RPC has been around for quite a while, it doesn't seem to get much love or respect from the WordPress community these days. Part of this could be because XML-RPC isn't the simplest of protocols: at the very least you need to understand and know how to manipulate XML. But another large part is likely how incomplete the API support is. It exposes certain WordPress operations, but it omits whole classes of things you might want to do remotely over an API. So what's a better option if we want to stick with the magical three letters "API"?

There are two choices. Once upon a time, Automattic, the commercial entity that runs the WordPress hosting at wordpress.com, made available a JSON-based REST service their customers could use. This was available for wordpress.com, but self-hosted WordPress sites couldn't use it. Later this functionality was added to their kitchen-sink plugin Jetpack (http://jetpack.me), which "supercharges your self-hosted WordPress site with cool functionality from WordPress.com." I've not used Jetpack on any site I've set up, largely because it always seemed a bit heavyweight to me even if it does do a ton of cool stuff simultaneously out of the box. Plus it introduces some dependencies on the wordpress.com backend infrastructure I didn't really want. That takes this option out of the running for me.

The second choice, better in some ways (worse in others, more on that in a moment), is a plugin that provides a similar JSON-based REST API. The later version of the plugin (v2, in beta) is meant to be a reference implementation merged into WordPress core in short order. This means the functionality will eventually be available out of the box without having to install a plugin. I'm not entirely sure if this is still the plan for WordPress roadmap, but the intent to add this to core is a pretty strong indicator of support. That's the good part of this option. There are two aspects that I am less enamored of: v2 of this plugin's implementation is relatively new, so information about installing and using it is much less mature than what is available for v1 (e.g., the API documentation at http://v2.wp-api.org is more a collection of section headings than actual documentation). This leads to lots of peeking back and forth between v1 and v2 docs and more hunting down of arcana/reading of the source than I would prefer. In this column, I will largely try to cut through all of that and provide some more direct instruction. There is, however, one place I'm going to punt on how to do things (my second negative); we'll come to that a little later on.

WP-API Install

Assuming again that you have a functioning and up-to-date WordPress install to work with, let's see how to get the WP-API stuff functional. There are 3–4 steps; let's start with the first two and bring the others in when we need them.

First off, you will want to install and activate the "WordPress REST API (Version 2)" plugin. You can either do this by entering that phrase into the search box in Plugins -> Add New (be sure to get the Version 2 one), or if you want to flex your dev chops, you can change to the wp-content/plugins directory of your WP installation and clone the plugin from its GitHub repo right into place:

```
git clone git://github.com/WP-API/WP-API.git
```

(Be sure to activate the plugin once you've installed it.)

The second step is to confirm you have a compatible permalinks scheme selected (Settings -> Permalinks in the dashboard). Any scheme except for the one listed as "Default" will work. Switch it away from Default to something else and save the change if this is not the case.

To confirm that the installation works, the v1 Getting Started guide (http://wp-api.org/guides/getting-started.html) suggests you can type the following:

```
curl -I {URL of your WP site}
```

The -I tells cURL to make a HEAD request because all we really need to see is the headers this returns. If everything is hunky-dory, you should see something like this:

```
$ curl -I http://local.wordpress.dev
HTTP/1.1 200 OK
Server: nginx
Date: Thu, 30 Jul 2015 03:17:23 GMT
Content-Type: text/html; charset=UTF-8
Connection: keep-alive
X-Powered-By: PHP/5.5.9-1ubuntu4.11
X-Pingback: http://local.wordpress.dev/xmlrpc.php
Link: <http://local.wordpress.dev/>; rel=shortlink
Link: <http://local.wordpress.dev/wp-json>; rel="https://
github.com/WP-API/WP-API"
```

The second Link: header we get back above is the key: it shows that WP-API is installed and ready to take requests at the wp-json endpoint. As a quick aside, the examples in this column will all be using a local WordPress install I have on my laptop provided by the Varying Vagrant Vagrants package (https://github.com/Varying-Vagrant-Vagrants/VVV). If you use Vagrant, be sure to check VVV out because it is quite well done.

Now That It's Installed, What Can We Do?

Now that we know it is working, what can we do with it? Let's actually ask it:

```
$ curl http://local.wordpress.dev/wp-json/
{"name":"Local WordPress Dev","description":"Just
another WordPress site","url":"http:\/\/local.wordpress
.dev","namespaces":["wp\/v2"],"authentication":[],"routes":{"
\/":{"namespace":"","methods":["GET"],"_links":{"self":"http:
\/\/local.wordpress.dev\/wp-json\/"}},"\/wp\/v2":{"namespace":
"wp\/v2","methods":["GET"],"_links":{"self":"http:\/\/local
.wordpress.dev\/wp-json\/wp\/v2"}},"\/wp\/v2\/posts":
{"namespace":"wp\/v2","methods":["GET","POST"],"_links":
{"self":"http:\/\/local.wordpress.dev\/wp-json\/wp\/v2\/
posts"}},"\/wp\/v2\/posts\/{id}":{"namespace":"wp\/v2"
,"methods":["GET","POST","PUT","PATCH","DELETE"]},"\/wp\/v2\/
posts\/schema":{"namespace":"wp\/v2","methods":["GET"],"
_links":{"self":"http:\/\/local.wordpress.dev\/wp-json\/
wp\/v2\/posts\/schema"}},"\/wp\/v2\/posts\/{parent_id}\/
meta":{"namespace":"wp\/v2","methods":["GET","POST"]},"\/
wp\/v2\/posts\/{parent_id}\/meta\/{id}":{"namespace":"wp\/
v2","methods":["GET","POST","PUT","PATCH","DELETE"]},"\/
wp\/v2\/posts\/meta\/schema":{"namespace":"wp\/
v2","methods":["GET"],"_links":{"self":"http:\/\/local
.wordpress.dev\/wp-json\/wp\/v2\/posts\/meta\/schema"}},
...
```

Note: I could have made this request via Perl (perhaps used GET from the LWP::Simple package, HTTP::Tiny, or any of the modules we've discussed in the past for this sort of thing) but cURL was already in my shell history.

Egads, that's one big blob of JSON we get back (I cut it off at an arbitrary point; the whole thing is 6443 characters total). It is kind of hard to read, so let's run it through a JSON pretty-printer to make it more legible. Again, we could write some Perl code to parse and pretty print, but in command-line cases like this, I tend to use one of two really great JSON tools: underscore-cli (https://github.com/ddopson/underscore-cli) or jq (http://stedolan.github.io/jq/). Both are excellent, so if you haven't encountered them before, I highly recommend you go check them out. Let's run that last request through jq (and show an excerpt from the reply):

```
$ curl -s http://local.wordpress.dev/wp-json/|jq .
{
  "name": "Local WordPress Dev",
  "description": "Just another WordPress site",
  "url": "http://local.wordpress.dev",
  "namespaces": [
    "wp/v2"
  ],
  "authentication": [],
  "routes": {
    "/": {
      "namespace": "",
      "methods": [
        "GET"
      ],
      "_links": {
        "self": "http://local.wordpress.dev/wp-json/"
      }
    },
    "/wp/v2": {
      "namespace": "wp/v2",
      "methods": [
        "GET"
      ],
      "_links": {
        "self": "http://local.wordpress.dev/wp-json/wp/v2"
      }
    },
    "/wp/v2/posts": {
      "namespace": "wp/v2",
      "methods": [
        "GET",
        "POST"
      ],
      "_links": {
        "self": "http://local.wordpress.dev/wp-json/wp/v2/posts"
      }
    },
    "/wp/v2/posts/{id}": {
      "namespace": "wp/v2",
      "methods": [
        "GET",
        "POST",
        "PUT",
        "PATCH",
        "DELETE"
      ]
    },
...
   "/wp/v2/users": {
      "namespace": "wp/v2",
      "methods": [
        "GET",
        "POST"
      ],
      "_links": {
        "self": "http://local.wordpress.dev/wp-json/wp/v2/users"
      }
    },
    "/wp/v2/users/{id}": {
      "namespace": "wp/v2",
      "methods": [
        "GET",
        "POST",
        "PUT",
        "PATCH",
        "DELETE"
      ]
    },
...
```

Let's take a closer look at some of this output. Specifically, I want to draw your attention first to the info it printed regarding the route available to query post info:

```
"/wp/v2/posts": {
  "namespace": "wp/v2",
  "methods": [
    "GET",
    "POST"
  ],
  "_links": {
    "self": "http://local.wordpress.dev/wp-json/wp/v2/posts"
  }
},
"/wp/v2/posts/{id}": {
  "namespace": "wp/v2",
  "methods": [
    "GET",
    "POST",
    "PUT",
    "PATCH",
    "DELETE"
  ]
},
```

This says I can either make a GET or a POST request for http://local.wordpress.dev/wp-json/wp/v2/posts to read or change the list of posts on the site. If I want to address an individual post (to GET, submit a new one with POST, DELETE it, and so on), I can do so at the same URL with the ID for that post tacked on to the path. This pattern repeats itself in the previous output for users, so we now know how to with users of the system. Let's try to get the list of users on the site:

```
$ curl -s http://local.wordpress.dev/wp-json/wp/v2/users|jq .
[
  {
    "code": "rest_forbidden",
    "message": "You don't have permission to do this.",
    "data": {
      "status": 403
    }
  }
]
```

Whoops, that didn't work—and good thing too! We really don't want anyone with cURL to be able to pull a list of users. That leads to the second part of the WP-API install/setup and a bit of a screed.

WP-API Authentication

In order for authentication of any type to work, there has to be an existing user defined on your site that you will authenticate to do the work. If you plan to query information that only an admin-level user should have access to (e.g., a list of site users), this user will have to be created as an admin. If you don't need that level of access from the API, I encourage you to create a user at a lower role or just send unauthenticated requests for publicly viewable information. New users for WP-API are created using the normal WordPress process (Users -> Add New). For this column, I created an admin user with the user name "api" and the password "api" (yup, security by alliteration, yay!) on my local test site.

WP-API has two contexts it operates in, one I'll call "internal," where code running on the site (e.g., a PHP-based WordPress theme), the other "external" (some outside code calls the API remotely). We're going to totally ignore the former and only look at the external context. In this context, there are two, maybe three mechanisms for authentication.

The first is the least secure one and is only recommended for development and testing. This is using the HTTP Basic Authentication found in RFC 2617. To use this, you need to git clone the Basic Authentication plugin into place as we did earlier:

```
$ cd wp-content/plugins
$ git clone git://github.com/WP-API/Basic-Auth.git
```

and then activate the plugin in the dashboard.

The second and third options are to use OAuth. OAuth is a mildly complicated protocol that comes in two incompatible versions (1.0a and 2.0) and that is designed to allow a third-party client to be given permissions to act on the behalf of a user. So, for example, if you install a new Twitter or Gmail client, it is likely that the first thing it will do is ask you to authenticate to those services and then permit that client to act on your behalf to perform certain operations (post tweets, manipulate your mail, etc.). This is OAuth in action.

Here's where it starts to get tricky and we quickly descend down a rabbit hole. The WP-API docs suggest that you install an OAuth 1.0a plugin from GitHub ("git clone git@github.com:WP-API/OAuth1.git content/plugins/oauth-server"; see https://github.com/WP-API/OAuth1) and use that for authentication. It is suggested that this plugin will also eventually be incorporated into WP core. Ordinarily at this point in the column, we'd go off and talk about how OAuth works and how to work with it in Perl. I won't be doing that here for two reasons:

1. The protocol/framework is a wee bit complicated and needs a little bit of explanation before you can dive into using it, and I don't have the space.
2. I'm annoyed that WP-API's suggested plugin implements 1.0a of OAuth and not 2.0. As far as I can tell, there isn't a major service provider using 1.0a instead of 2.0, so the value of going deeper into a barely used protocol is unclear to me. Some say that the older version was a stronger protocol, but I'm not sure that pragmatically justifies the column space.

Now let me make things even more interesting. There exists a commercial plugin (or at least one that would like to charge licensing fees) that implements OAuth2. It can be found at https://wp-oauth.com. It claims to support WP-API (at least in part of the doc, while in another part it claims it doesn't, sigh). I'm also not clear whether it supports the 2.0 WP-API beta version either. Because OAuth2 leans on SSL/TLS for some of its security, you would want that set up on your site before truly using it. I have yet to test it.

Given these complications, I'm going to punt on the more secure methods (even though I know it means that somewhere an angel isn't going to get its wings) and just go with Basic Authentication in our examples. Just so you don't feel I'm hanging you out to dry, I will mention that the Net::OAuth and Net::OAuth2 modules (plus a couple others like OAuth::Lite) do exist, so you can definitely perform OAuth operations (from both protocol versions) from Perl. If you'd like to see another column about just OAuth, please drop me a line and I will see about writing one.

To review as we leave the rabbit hole: to use WP-API operations of a certain level, you need a suitably empowered WordPress user and a way to authenticate as that user installed. We'll be using the Basic Authentication plugin for the latter (boo hiss).

Perl Time

In a previous column about using REST interfaces from Perl, we tiptoed up to using Perl modules that provided lots of "do what I mean" syntactical sugar. In this column, I'm just going to put the pedal to the metal and go right for using that kind of module.

Let's start off with getting the list of pages on a site. Our first attempt to write code to this would probably look a bit like this:

```
use WebService::CRUST;

my $w = new WebService::CRUST(
   base      => 'http://local.wordpress.dev/wp-json/wp/v2/',
   format    => [ 'JSON::XS', 'decode', 'encode', 'decode' ],
);

# this is the equivalent of
# $w->get('pages');
# we could also write $w->pages;
# yummy syntactic sugar!
my $result = $w->get_pages;

print "Total items: " .
  $result->crust->{response}->{_headers}->{'x-wp-total'},
  "\n";

foreach my $page ( @{ $result->result } ) {
      print $page->{'id'} . ':' .
            $page->{'title'}->{'rendered'} .
            " (" . $page->{'link'} . ")\n";
}
```

The code creates a WebService::CRUST object and tells it that all of our requests are going to start with that URL. It also specifies that we will want to use JSON::XS (the faster JSON parser) to decode the responses we get back. The next step is to query for all of the pages on the system. As you can see in the comments, WebService::CRUST allows us to write code that makes it look like pages() or get_pages() is a real method call. This is one of the things I like about this module: it makes for very readable code, even if it is doing a bit of autoload magic behind the scenes.

For fun (or actually, for foreshadowing), we reach deep into the WebService::CRUST::Response object using the crust method to pull out one of the headers WordPress sends us in response to our query (X-WordPress-Total, which gets downcased when stored in the object). This header provides the number of items we should expect back from our query. Then we proceed to iterate over the response we got back in that WebService::CRUST::Response object (via the result method) to print out the ID, title, and the URL for each page on the system. Here are the results on my local test instance (which I've preloaded with a bunch of example pages):

```
Total items: 248
2:Sample Page (http://local.wordpress.dev/sample-page/)
5434:2008 Festival (http://local.wordpress.dev/archive
/2008-festival/)
5433:2007 Festival (http://local.wordpress.dev/archive
/2007-festival/)
5432:2006 Festival (http://local.wordpress.dev/archive
/2006-festival/)
5409:2012 Festival (http://local.wordpress.dev/archive
/2012-festival/)
5407:2011 Festival (http://local.wordpress.dev/archive
/2011-festival/)
5405:2010 Festival (http://local.wordpress.dev/archive
/2010-festival/)
5403:2009 Festival (http://local.wordpress.dev/archive
/2009-festival/)
```

```
5305:Pick-Up Band 2014 (http://local.wordpress.dev/archive
/2014-festival/pick-up-band-2014/)
5280:Saturday Schedule by Location (2014) (http://local
.wordpress.dev/archive/2014-festival/saturday-in-davis-square
/saturday-schedule-by-location-2014/)
```

Hey, wait a second, something is wrong. WordPress says there are 248 pages on the system, but it has only returned 10. Welcome to the world of pagination. Perhaps showing its blogging roots, WordPress wants to hand back the reply one "page" at a time. This isn't entirely out of the ordinary because other servers (e.g., LDAP servers) often have a max size for data returned that you can only deal with by requesting a chunk at a time. We could try to get around this pagination by asking WordPress to create pages that are big enough to hold all of the items or even turn off pagination, but I think it is better to work within the system than try to hack around it.

So how do we get more pages past the first one? If we were to peek more closely at what was returned from our request, we would notice that WordPress has sent us a "link" header (remember that from the beginning of the column?). Here's what it looks like from the request above (it is all one long line):

```
'http://local.wordpress.dev/wp-json/wp/v2/pages?page=
2>; rel="next"'
```

That is the URL we will have to request to get the next set of results (i.e., the next page). We'll need to write code that parses this header and extracts the next page number, then repeats the request. Here's what that code looks like:

```
use WebService::CRUST;

my $w = new WebService::CRUST(
   base     => 'http://local.wordpress.dev/wp-json/wp/v2/',
   format   => [ 'JSON::XS', 'decode', 'encode', 'decode' ],
);

my $nextpage = 1;

my $result = $w->get_pages( 'page' => $nextpage, );

print "Total items: " .
     $result->crust->{response}->{_headers}->{'x-wp-total'},
     "\n";
print "Total pages of content: " .
      $result->crust->{response}->{_headers}->{'x-wp-totalpages'},
      "\n";

while ( defined $result and $nextpage ) {
   foreach my $page ( @{ $result->result } ) {
      print $page->{'id'} . ':' .
           $page->{'title'}->{'rendered'} .
           " (" . $page->{'link'} . ")\n";
   }

   ($nextpage) =
    $result->crust->{response}->{_headers}->{'link'} =~
    /\?page=(\d+)>; rel="next"/;

   last unless ( defined $nextpage );

   $result = $w->get_pages( 'page' => $nextpage, );
}
```

Let's focus for a moment on how this differs from the previous code. WordPress is willing to tell us how many pages it will take to provide the entire result set, so I print that for informational purposes. For the real work, our get_pages requests now take an argument that is the parameter and the value to be sent with that request. Adding this argument means we'll be requesting:

```
http://local.wordpress.dev/wp-json/wp/v2/pages?page=N
```

where N is the value of $nextpage. We print the information returned for that page, determine if there are more pages (as specified in the link header), and if so, we perform another request for the next page. As a quick aside, we could have taken the number of pages returned in the X-WP-TotalPages header and iterate from page 1 to that value, but I believe it is less likely to cause a race condition if we work from the "here's the next page" info we get back on each query instead.

This is the basic pattern for most things we can pull back from the API. For example, if we wanted a list of users:

```
use WebService::CRUST;
my $w = new WebService::CRUST(
   base     => 'http://local.wordpress.dev/wp-json/wp/v2/',
   basic_username => 'api',
   basic_password => 'api',
   format   => [ 'JSON::XS', 'decode', 'encode', 'decode' ],
);

my $nextpage = 1;

my $result = $w->get_users( 'page' => $nextpage, );

while ( defined $result and $nextpage ) {
   foreach my $user ( @{ $result->result } ) {
      print $user->{'id'} . ':' . $user->{'name'} . "\n";
   }

   ($nextpage) =
    $result->crust->{response}->{_headers}->{'link'} =~
    /\?page=(\d+)>; rel="next"/;

   last unless ( defined $nextpage );

   $result = $w->get_users( 'page' => $nextpage, );
}
```

Almost identical, yes? The only differences are that we add some initial parameters to send along authentication with every request (in an insecure manner, don't rub it in) and later on pull out different fields from the returned information.

Where Do We Go from Here?

We're almost out of room, but there are a few important things to mention. First off, the examples we've seen here all pull a collection of items (pages, users, etc.). If we want to retrieve a single item, we can reference that item as a part of the path we request by appending the ID we need—for example:

```
    $result = $w->get( "pages/$id" );
```

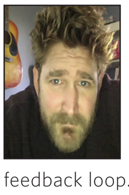
Second, we've only seen examples that retrieve content. If we want to create or modify content on the site, we use the REST idea of using other HTTP operations as verbs. Want to create a new page or edit a page? Perform a PUT request (details found in the v1 documentation) with the right parameters specified as arguments to the put() method.

And lastly, one more advanced topic we didn't discuss is how to use more of the power of WordPress in our interactions. v1 of the API had a working "filter" parameter which allowed you to pass in a specification the WordPress WP_Query class could work with. This means that you could say to WordPress "return all of the posts by this author" or "only return a list of publicly published posts." I had difficulty using this facility with the v2 API because I believe it is still very much a work in progress as of this writing. Hopefully, this facility will be up to snuff by the time you read this.

In the meantime, enjoy, and I'll see you next time.

iVoyeur
Using NCPA: Nagios Cross-Platform Agent

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

I've been working at home for over a year now, and I can't help but feel that I'm somehow doing it incorrectly. I'm wearing pants for one thing, and my hygiene habits have not changed whatsoever (although admittedly I never was a hygiene Olympian). In fact I seem to be experiencing very few of the great benefits one hears about, like drinking at inappropriate times, playing video games, not interacting with people, and, well, not working.

In their place I'm experiencing a whole slew of not very awesome side effects of having these large, luxurious blocks of uninterrupted time to dig in and work on stuff. These include failing to stop working ever and starting too many side-projects because of all the "extra time" I feel like I have (that I don't actually have). I even have meetings. Oh crap, in fact I have meetings right now; I'll be right back.

Okay sorry, that's another problem: meetings sneak up on me now, and nearly always coincide with one meal or another that I'm supposed to be eating. An unlucky consequence, I suppose, of the dissonance between the people in my life who make meetings and live two hours ago, in California, and the people in my life who make lunch and dinner and who live now, in Texas. It also has begun to seem weird that we have times for these things at all, eating, meeting, and working, that is.

When I applied for this job, my first several interviews were undertaken by way of Google Hangouts. This was a very real logistical concern for me at the time because I was running a snowflake everything-compiled-from-scratch Linux laptop, and, well, you know how that goes with cameras, soundcards, printers, and etc. I got it all sorted out in time, and experienced my first few video-chats as job interviews, which, by the way, is not a very good idea. It was extremely awkward and I kept spacing out. It felt like I was watching a job interview on TV, so I kept forgetting to answer.

Anyway, I spend an inordinate amount of time on Hangouts, appear.in, and various other hosted impromptu meeting services these days, and I've noticed that whenever Hangouts is going, my CPU fans kick on. This is pretty noticeable on my MacBook, but downright distracting on my ThinkPads. My poor little ThinkPad x120 gasps and wheezes like it's sprinting the last 30 feet of an ultra-marathon when I try to run Hangouts on it.

Being a monitoring sort of person, I got curious about this behavior, and brought some tools to bear to help me visualize the overhead, but I pretty quickly got myself entangled in the question of whether I was comparing apples to apples. I mean literally. Is it the same thing to measure CPU utilization on an Apple vs. a Linux box?

At this point I should point out that not only am I lazy by nature, but I also really don't have the time to put any actual effort into this, so I figured the shortest path was probably to get my hands on a cross-platform monitoring agent. That would at least make me feel like I was measuring both systems with the same ruler, and that'd probably help me to brute-force ignore the screams of protest from my inner engineer.

Because I spend my time these days thinking about and working with telemetry processing systems, I haven't really looked at the state of client-side data collection tools lately (especially tools that might work on a Mac). There aren't many cross-platform monitoring agents that include support for OS X. The most robust solution is probably DataDog, but that was overkill for my purposes. I wanted something I could use for a few days and then get rid of, and setting up DataDog would entail … artifacts like emails, and passwords on Web sites, and well-intentioned pre-sales, and support representatives.

Really, I just wanted something like good-old GKrellM, so I spent a few minutes trying to get GKrellM to build on my Mac, which was fun but fruitless. I was also a little surprised to find there was no homebrew recipe for GKrellM; "brew install X" so rarely fails me nowadays. Then I remembered NCPA.

NCPA, or the "Nagios Cross-Platform Agent," is a monitoring agent built and maintained by the folks at Nagios Enterprises. It's a cross-platform Python script that is distributed in binary form (via cx_Freeze). In many ways, it's exactly what you'd expect if you asked Nagios Enterprises for an agent. It's small, easy to work with, and, out of the box, it doesn't really know how to monitor very much of anything. It can enumerate the running processes and measure CPU, Memory, Disk, and Network utilization. And it does a great job of detecting all of these things (it sees all of my vmnet network interfaces, for example), but like its big brother it depends on plugins to do the heavy lifting, and that's a good thing IMO.

I'd never tried NCPA, so I thought this would be a great opportunity. It, along with Nagios Core and the rest of the open source software made by Nagios Enterprises, is on GitHub. I must be getting old, though, because I just went and grabbed the official binary distributions of NCPA for OS X and Debian from [1]. The Linux install was pretty much what you'd imagine: one dpkg-i and it was up and running.

The Mac put up a little more resistance. NCPA came packaged in a disk-image (.dmg file), which contained an installer shell script called install.sh. I could not chmod the script to make it executable because .dmg's are a read-only file system. All of my

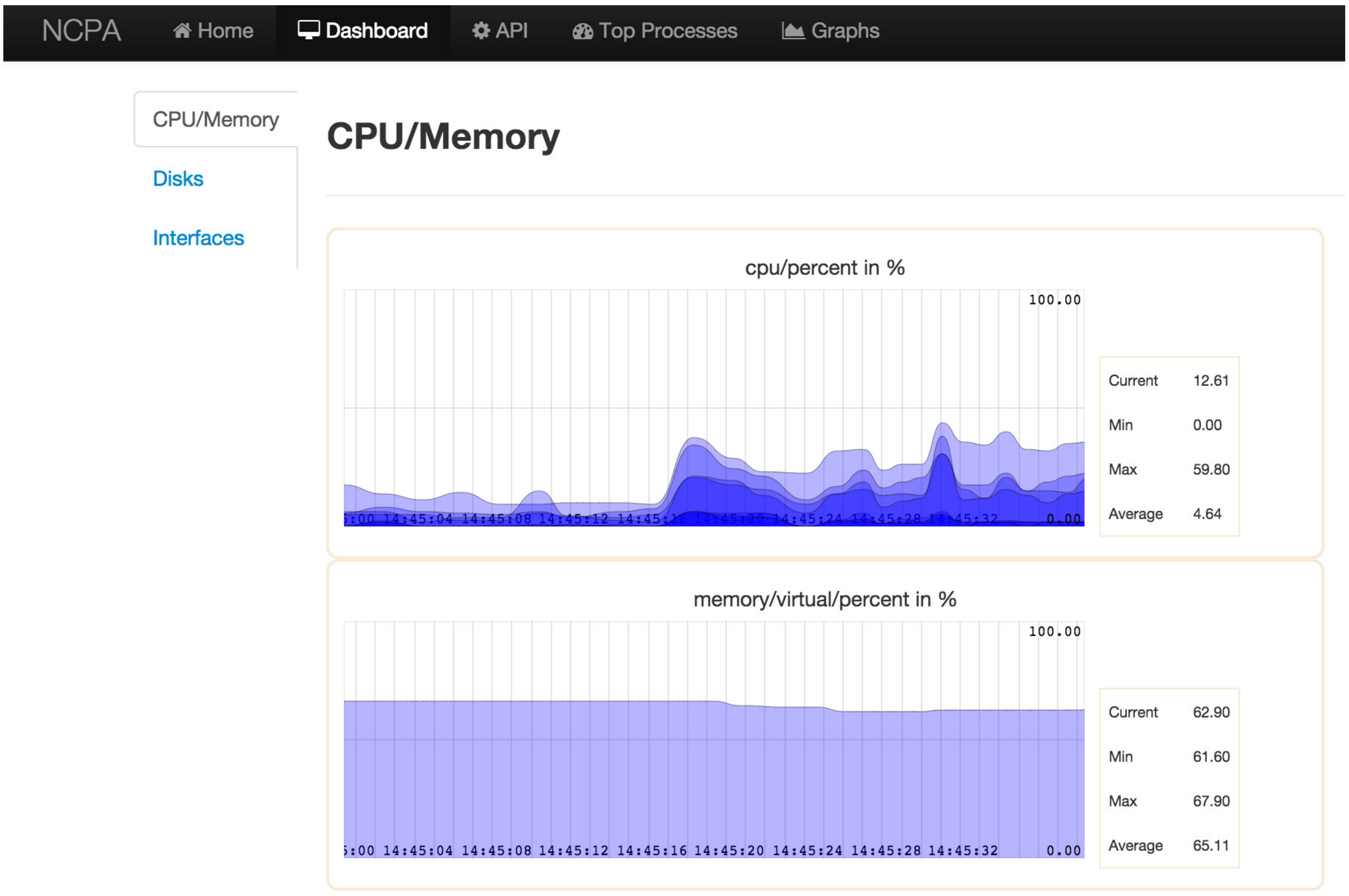


Figure 1. NCPA's spartan but functional built-in Web interface

attempts to remount it as R/W were mockingly rejected by OS X. Giving up on that, my first few attempts to indirectly execute the script with, for example, "sh -c" were also rebuffed, but...

```
/bin/sh < /Volumes/NCPA-1.8.1/install.sh
```

...worked for me.

Like Nagios, NCPA installs itself to /usr/local/ncpa by default. Inside this directory is an "etc" where you will find an ncpa.cfg file that controls NCPA's behavior. I left most of this alone, changing only the "community_string" attribute, which specifies the auth token you use to interact with NCPA.

Compared to the other agents, like Check_MK, commonly used in the Nagios solar system, NCPA is a lot easier to install and reason about. It eschews custom protocols and provides a Web-API that responds in JSON over HTTPS on tcp/5693 by default (change this along with everything else in the config file). This is pretty great. You can interact with NCPA using cURL or anything else that can speak HTTPS, and you can parse its output with jq, or anything else that groks JSON.

It even comes with a Web UI that draws graphs!

Granted, it's missing some fundamental features that I look for in a metrics analysis tool. Its y-axis handling leaves a lot to be desired, for example, but the UI is fine for ad hoc checking out individual boxes, and obviously it was more than sufficient for my current purposes. Anyway, NCPA really isn't here to be an analysis tool; it's a lightweight, easy-to-run data collection agent. One that, if I were in the market for an agent, is actually a quite compelling choice.

I mean look at this API! There are eight top-level URIs: memory, interface, agent, CPU, disk, agent, process, and services. I can, for example, get a JSON dump of the running processes on my MacBook at

```
https://localhost:5693/api/processes/
```

I can get the free memory with

```
https://localhost:5693/api/memory/virtual/available
```

I'm oversimplifying just a tad there. If you're doing this outside of a browser, you'll need to pass in the token by setting it as an attribute in the URL like so:

```
https://localhost:5693/api/memory/virtual
/available?token=zomgsecret
```

There are a slew of other attributes we can set: for example, get Nagios-style output by setting threshold attributes like so:

```
https://localhost:5693/api/memory/virtual/available?token
=zomgsecret&warning=1&critical=2&check=true
```

If you copy or symlink some standard Nagios plugins into /usr/local/ncpa, you can even run them from the API from the agent tree like so:

```
https://localhost:5693/api/agent/plugin/check_thing
/"First Arg"/"Second Arg"/?token=zomgsecret
```

You'll get back a JSON blob of the plugin's output that looks like this:

```
{ "value": { "returncode": 0, "stdout": "Thingy looks ok! First
Arg, Second Arg\n" } }
```

If you aren't already using check_mk and especially if you're running NRPE/NRDP, then you might want to consider running NCPA as a replacement for your current remote plugin-execution framework. In my admittedly teensy experience, it's been simple and painless, and has a slew of features built in for emitting to preexisting NRDP daemons and otherwise cohabitating with your existing Nagios toolchain.

It certainly scratched my itch for comparing the utilization characteristics of the various video conferencing tools I use every day (for the moment, it looks like appear.in on my MacBook is the best option). The next time I'm helping someone design and/or build out their Nagios infrastructure, NCPA will definitely play a role.

Take it easy.

Resources

[1] NCPA agent: https://assets.nagios.com/downloads/ncpa/download.php.

For Good Measure
The Denominator

DAN GEER

Dan Geer is the CISO for In-Q-Tel and a security researcher with a quantitative bent. He has a long history with the USENIX Association, including officer positions, program committees, etc. dan@geer.org

Let's start with a recent paper that is very much worth your time to read: "Global Cyberspace Is Safer than You Think: Real Trends in Cybercrime" by Eric Jardine and released by Chatham House this past July [1]. Its message is exactly that given by its title: that cyberspace is getting better—not getting worse, that cyberspace is getting more safe—not getting more dangerous. The argument for that message is that thinking cyberspace is ever worse, ever more dangerous comes from failing to properly normalize whatever measures of safety you've heretofore been paying attention to. It is only fair to quote its front matter directly:

> Information technology (IT) security firms such as Norton Symantec and Kaspersky Labs publish yearly reports that generally show the security of cyberspace to be poor and often getting worse. This paper argues that the level of security in cyberspace is actually far better than the picture described by media accounts and IT security reports. Currently, numbers on the occurrence of cybercrime are almost always depicted in either absolute (1,000 attacks per year) or as year-over-year percentage change terms (50 percent more attacks in 2014 than in 2013). To get an accurate picture of the security of cyberspace, cybercrime statistics need to be expressed as a proportion of the growing size of the Internet (similar to the routine practice of expressing crime as a proportion of a population, i.e., 15 murders per 1,000 people per year)....In particular, the absolute numbers tend to lead to one of three misrepresentations: first, the absolute numbers say things are getting worse, while the normalized numbers show that the situation is improving; second, both numbers show that things are improving, but the normalized numbers show that things are getting better at a faster rate; and third, both numbers say that things are getting worse, but the normalized numbers indicate that the situation is deteriorating more slowly than the absolute numbers. Overall, global cyberspace is actually far safer than commonly thought.

In short, Jardine is saying that the denominator matters, i.e., that reporting counts of anything is poorer decision support than reporting rates and proportions, that counts of events per unit time will, and must, mislead. It is incorrect to talk about how much mayhem there is without talking about how much opportunity for mayhem there is.

Jardine's line of critique is entirely straightforward, and cyberspace is not the only place that such arguments about the validity of inference are taking place. As a prominent example, consider Stephen Pinker's 2012 book *The Better Angels of Our Nature: Why Violence Has Declined*. In a synopsis in the *Wall Street Journal*, he wrote:

> We tend to estimate the probability of an event from the ease with which we can recall examples, and scenes of carnage are more likely to be beamed into our homes and burned into our memories than footage of people dying of old age. There will always be enough violent deaths to fill the evening news, so people's impressions of violence will be disconnected from its actual likelihood.

This is, again, an argument for looking at rates and proportions rather than counts. But in a direct cross, Nassim Nicholas Taleb responded with a paper, "On the Super-Additivity and Estimation Biases of Quantile Contributions" [2], in which he argues that when a distribution is fat-tailed, estimations of parameters based on historical experience will inevitably mislead:

> When I finished writing *The Black Swan*, in 2006, I was confronted with ideas of "great moderation," by people who did not realize that the process was getting fatter and fatter tails (from operational and financial leverage, complexity, interdependence, etc.), meaning fewer but deeper departures from the mean. The fact that nuclear bombs explode less often than regular shells does not make them safer. Needless to say that with the arrival of the events of 2008, I did not have to explain myself too much. Nevertheless people in economics are still using the methods that led to the "great moderation" narrative, and Bernanke, the protagonist of the theory, had his mandate renewed.

And to highlight his central point:

> [We are] undergoing a switch between [continuous low grade volatility] to … the process moving by jumps, with less and less variations outside of jumps.

You will possibly find Taleb's paper difficult, but he is speaking to our interest in cybersecurity—are we getting worse or are we getting better? Is there anything we are currently measuring that is leading us to conclude that we are doing the right thing(s) as inferred from measurements of what we believe to be outcomes? Are our inferences confounded with little understood assumptions about thin tails (Gaussian) when we are actually in a fat-tailed (power law) situation? Are we moving into a world where, as Taleb suggests, we are switching from continuous low grade volatility to less frequent but much larger jump changes in the state of the (our) world?

The present author asked this question in a naive form in the spring of 2008 at SOURCE Boston:

Everyone but everyone classifies the 9/11 attack as out-of-nowhere—a black swan to use Taleb's terminology. That attack changed everything because it was not foreseen. It was a physical attack, but we, here, deal in digital attacks. Many of us have heard the phrase "Digital Pearl Harbor," and many of us here have wished we hadn't. If we talk with a member of the general public, we are likely to hear something like, "Look, you paranoid worry-warts keep predicting a big problem and if it was all that likely it would have happened by now. In fact, every day that goes by without something like that happening makes it more likely that it never will. Would you just stop bothering me?"

Now, a statement like, "That we have gone this long without anything big happening" is precisely the kind of statement that expects stability to continue, and which is necessary but not sufficient for a punctuation of that stability. If we look at 9/11 as digital security people, we might remember that the Nimda virus appeared the evening of September 18, 2001, i.e., a week later. Until that point, we'd never seen a virus that had carried more than one method of attack, and Nimda had five. So, to begin with, even if we had known everything about each of those five methods, including population statistics for the numbers and connectivity of vulnerable machines, we would not have predicted the ability of Nimda to spread as it did as we had not yet thought to model the union of multiple vulnerabilities.

That, however, is not all. For writers of classic virus attacks, the measure of their success is the energy differential between the first entry into a given target and the second, i.e., the bigger the difference in how hard it is to break in the first time and how easy it is to break in the second time, the bigger the win. The lowest energy way to maximize this energy differential is to install a new back door. Nimda followed this pattern and installed such a back door.

Because it had five methods for propagation and because it was evidently written with speed in mind, Nimda was also the fastest spreading virus we had yet seen. That rate of spread is known among infectious-disease people as virulence, and we'll return to that in a moment.

As you know, nearly all malware in the wild persists there. An older virus called E911 was such an example. E911 would cause your modem to dial 911 repeatedly; that is all it did. Now when I call you on the phone, the circuit stays up until the calling party disconnects. When I call 911, however, the circuit stays up until the called party disconnects, a difference that is done at the switch for the obvious reason that you do not want the intruder to cut the phone line and the police dispatcher to have to say, "Now whom was I talking to?" For the police to hang up on a 911 call when the calling party has gone away requires a human decision, made under uncertainty, done at human time scales. Because of this, it is possible to saturate a 911 console and that is precisely what the E911 virus was crafted to do.

The E911 virus was old and forgotten on September 18, 2001, but it was still available on the Net, and, of course, the Internet in the fall of 2001 was still dominated by dial-up connections. We got lucky in the simplest, stupidest, dumb luck kind of way. No jackass had the imagination to grab the E911 virus and re-target it at the back door Nimda was busy installing at warp speed everywhere while we all were preoccupied with watching CNN 24x7. If someone had done that, then everyone in America would have gotten up the morning of September 19 only to find that there was no emergency service available nationwide; it would have been turned off everywhere and all at once, like a light switch. While that would not have been a disaster of a physical sort, I submit that it would have been a grand mal seizure of the public confidence. Clinically, that defines terror; it would have required no planning just opportunistic reaction, and it would have been an unpredictable event whose downstream influence was out of all proportion to its concrete effects. It would parallel the Treasury's position that money lost or banks folded is a private tragedy of no importance, but that public loss of confidence in the financial system must be avoided.

On September 18, 2001, we escaped a public loss of confidence by luck and luck alone. As such, the next time someone tells you that the absence of a major Internet attack to date makes the absence of one tomorrow more assured, you can remind them that this proof (that we escaped such an attack by dumb luck) puts to bed any implication that every day without such an attack makes such an attack less likely. It does not make it less likely, but what it does most assuredly do is make it more surprising when it does come.

So is cyberspace getting worse or getting better? Jardine asks us to normalize how many events did occur to the size of how many events could have occurred, not how many did occur in an interval of unit time. He is correct that the possible event space is expanding dramatically, accelerating in its expansion by all accounts. Part of that is network extent, which I've estimated as having a 35% compound annual growth rate [3]. Part of that is the question of attack surface, per se [4]. In any case, Jardine is right that when we count events, we are misleading ourselves as to whether we are getting better or getting worse. But does changing the divisor alone really make the correction we need?

There is a power law here, to be sure. Wikipedia's concise reminder (under "Power Law") is that "Power-laws have a well-defined mean only if the exponent exceeds 2 and have a finite variance only when the exponent exceeds 3; most identified power laws in nature have exponents such that the mean is well-defined but the variance is not, implying they are capable of black swan behavior." That, my friends, is our situation—cyberspace does not have a well-defined variance for what can go wrong and hence cyberspace is unarguably capable of black swan behavior.

Elroy Dimson famously suggested that the definition of risk is that "more things can happen than will happen" [5], and our rate of growth in interdependence is absolutely making the number of things that can happen larger. Unfortunately, complexity prevents us from counting the number of things that can happen, and hence Jardine's argument that we divide the number of things that did happen by the number of things that could have happened is correct in spirit but would be irrelevant if our estimate of the number of things that could have happened were to be wrong.

Yet if the denominator is the number of things that could have happened and we severely underestimate that, doesn't that make the news even better? Taleb says "no" emphatically; the fat tails of power law distributions enlarge the variance of our estimates, leading to less frequent but more severe failures (The Black Swan). The best one could say is that most days will be better and better but some will be worse than ever. Everything with a power law underneath has that property (think earthquakes and whether one is overdue in California), and cyberspace's interconnectivity and interdependence are inherently power law phenomena.

Put differently, are pessimists getting the right answer for the wrong reasons? Is what Pinker said about the memorableness of televised violence making violence seem more prevalent than it is both true and yet misleading? Is what Jardine said about how looking at time series of cybersecurity failures is inherently misleading when the numbers of failures are not normalized in some way? Is what Taleb describes as the trivializing of risk when a power law is mistaken for a Gaussian the heart of what is in play?

In an article in the *San Francisco Chronicle* [6], Thomas Lee recounted how

> I found myself at a dinner in a fancy Menlo Park hotel to discuss cybersecurity with the executives of top Silicon Valley firms. [T]he mood was decidedly grim.
>
> "A devastating cyberattack is likely to occur in the next five years," said a top HP exec. Companies are nowhere near prepared for it. Neither are the feds. There were plenty of comparisons to hurricanes and earthquakes.
>
> "A slow-moving train wreck," one executive said.
>
> There [is] a kind of collective cognitive dissonance in Americans' thinking about tech. We'll eagerly pursue new innovations like the Internet of Things and electronic health records even as we're increasingly aware how vulnerable such technology makes us to terrorists and criminals.
>
> What struck me about the dinner, attended by executives from Hewlett Packard, Cloudera and PayPal, along with academics and investors, was the naked pessimism expressed by those in the room. Nobody even tried to put a happy face on the situation.

Are those executives, academics, and investors getting the right answer for the wrong reasons? Are Jardine and/or Pinker getting the wrong answer for the right reasons? Is it a truism that when risk cannot be estimated it will therefore be underestimated [7]? How do we tell if we are getting better or getting worse, and how can we explain this to citizens, regulators, and reinsurers? Is Taleb right that fat-tailed distributions and asymmetry are where risk lies and, which is more, that the apparent suppression of small failures is "balanced" by yet-to-be-observed black swan excursions?

Resources

[1] Eric Jardine, "Global Cyberspace Is Safer than You Think: Real Trends in Cybercrime," Centre for International Governance Innovation, Chatham House: http://www.cigionline.org/sites/default/files/no16_web_0.pdf.

[2] Nassim Nicholas Taleb, "On the Super-Additivity and Estimation Biases of Quantile Contributions": http://www.fooledbyrandomness.com/longpeace.pdf.

[3] Dan Geer, "T. S. Kuhn Revisited": http://geer.tinho.net/geer.nsf.06i15.txt.

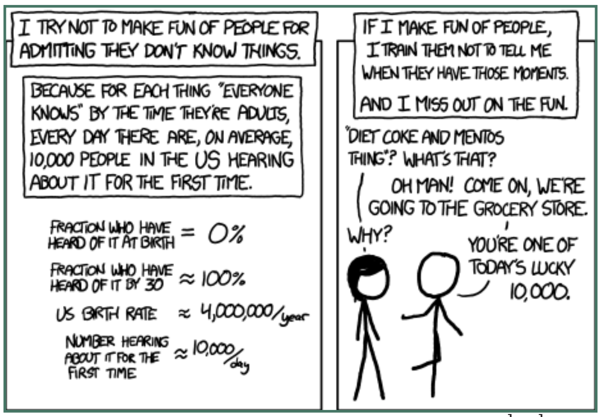
[4] Dan Geer, "Attack Surface Inflation": http://geer.tinho.net/geer.secot.7v14.txt.

[5] Peter L. Bernstein on Risk (Flash video): http://www.mckinsey.com/insights/risk_management/peter_l_bernstein_on_risk.

[6] Thomas Lee, "Forget Target, Ashley Madison Hacks; a Bigger Online Threat Looms" (paywalled): http://www.sfchronicle.com/business/article/Forget-Target-Ashley-Madison-hacks-a-bigger-6395645.php.

[7] Dan Geer, "Cybersecurity as Realpolitik": http://geer.tinho.net/geer.blackhat.6viii14.txt.

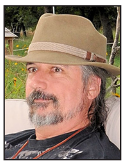
XKCD


I TRY NOT TO MAKE FUN OF PEOPLE FOR ADMITTING THEY DON'T KNOW THINGS.
BECAUSE FOR EACH THING "EVERYONE KNOWS" BY THE TIME THEY'RE ADULTS, EVERY DAY THERE ARE, ON AVERAGE, 10,000 PEOPLE IN THE US HEARING ABOUT IT FOR THE FIRST TIME.
FRACTION WHO HAVE HEARD OF IT AT BIRTH = 0%
FRACTION WHO HAVE HEARD OF IT BY 30 ≈ 100%
US BIRTH RATE ≈ 4,000,000/year
NUMBER HEARING ABOUT IT FOR THE FIRST TIME ≈ 10,000/day
IF I MAKE FUN OF PEOPLE, I TRAIN THEM NOT TO TELL ME WHEN THEY HAVE THOSE MOMENTS. AND I MISS OUT ON THE FUN.
"DIET COKE AND MENTOS THING"? WHAT'S THAT?
OH MAN! COME ON, WE'RE GOING TO THE GROCERY STORE.
WHY?
YOU'RE ONE OF TODAY'S LUCKY 10,000.

xkcd.com

/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

I am not a computer scientist. I don't designs 'em, I just runs 'em. When talk 'round the cocktail party buffet table turns to B-trees, linked lists, and why bubble sorts are awful, I grab another stuffed jalapeño and look for a less esoteric knot in which to mingle. Consequently, I am in no position to pontificate with authority or even basic coherence on any topic that contains a CS-related word more advanced than "algorithm." I had probably been a sysadmin for ten years before I learned what *that* one meant.

Computer science and systems administration are very different disciplines. Whenever I see a job opening for a system administrator that lists as one of its requirements an undergraduate degree in computer science, I roll my eyes and write that company, or at least their HR department, off as *personae sans clue*. Requiring a CS degree for your sysadmins is like insisting a Formula 1 driver possess a degree in traffic engineering. It's essentially a non sequitur. I think I've ranted on this before.

We used to joke that the only truly secure system was one with no I/O devices that had been encased in concrete and dumped into the Challenger Deep. Apparently that was no joke. Recent events have shown us that any system connected to another not only *can* but eventually *will be* compromised. I would now go so far as to say if you have *ever* used a credit card, applied for a US security clearance, or shopped online, your information is available to anyone who cares to purchase access to it. You and I and virtually everyone you know have been, to bring the subject into sharp focus, quite thoroughly pwned.

Prior to becoming a full-time writer I made my career, such as it was, in information security. Back in those days we naively believed that, were proper precautions taken and best practices followed faithfully, you could operate an enterprise-level network in relative safety where the vaunted C-I-A (confidentiality, integrity, and availability) were concerned. It's become increasingly obvious over the past few years, however, that networking is rotten at its most fundamental core, security-wise, and can't be fixed. I think the only sensible way to proceed from this point forward is to assume that every single bit of data you place in any networked environment will be, without any realistic possibility of sanctuary, compromised.

My personal solution, were money and profound inconvenience no obstacles, would be to tear the entire network infrastructure down and start over again from square one. In my idealized network protocol, which I will call the "No-Eavesdropping Data Transfer Protocol," or NEDTP, all connections would be point-to-point and determinative, meaning every device knows for an indisputable fact the identity of the other devices to which it is attached. No spoofing is possible. No man-in-the-middle attacks are possible. When a packet comes in, its origin and data integrity are assured by the simple expedient that every link along the way is known and any tampering modifies the integrity hash in an irreversible manner. This would suck from a privacy standpoint, but what we have now isn't exactly exemplary in that department. At least in my world you could buy crap online without needing your phone in the other hand to cancel that account when, moments later, the first inevitable fraudulent charge came through.

How easy would this be to implement? Beats me. I'm just the idea guy here. If I really had an unhackable network protocol, I'd be rubbing elbows with Elon Musk and having craters on Charon named after me. I'm just a humorist, after all. But I do know *something* needs to be done. I'm tired of getting emails and/or snail mail letters every other week announcing that yet another of the supposedly secure data archives to which my life's statistics are entrusted has been breached by hackers working for career criminals or the unfriendly state du jour. I have more free subscriptions to *LifeLock* these days than pairs of wearable shoes.

Maybe we should just stop *trying* to protect our vital information. If we all simply proceed on the presumption that every purchase, every bank account, every electronically enabled transaction of any sort is being monitored by criminals who fully intended to exploit whatever information we provide in order to conduct them, we might actually be safer. Instead of established account numbers, we could all use one-time pads that ceased to be of further utility the moment they were used. That pretty much describes all of my credit accounts, anyway.

I'd like to devote the rest of this column to addressing the thorny issue of operating systems, specifically the requirement thereof. Way back in the Cretaceous era of computing someone decided that just having a computer wasn't enough: it needed to be *usable* for something. So long as all users were engineers and deeply competent in machine language, the usability monster did not dare raise its misshapen head, but once the proposal was put forth that people who were not married to the system might want to do computing as well, the toxin-belching chimera was released into the wild.

It became obvious before long to those tasked with implementing this radical idea that some form of interface between the human who spoke a rich language full of nuance and complex syntactical rules and the machine that only made use of ones and zeroes was going to be needed. Various solutions were suggested to fulfill this requirement, the battles among the various camps reaching epic proportions at times. After much acrimony and several spoiled friendships, the basic operating system design that we know and love/loathe today emerged victorious.

What alternatives do we have to the familiar architecture? I, for one (because multiple personalities are so far not one of the mental aberrations with which I am saddled), would prefer that operating systems be stripped down. The examples we have today are so bloated with "features" that people spend huge chunks of their careers just trying to understand them. That's messed up, if you think about it. These are devices that are supposed to simplify our lives so we can devote more time and attention to the stuff that really matters, not occupy vast tracts of neocortical real estate in and of themselves.

Imagine needing a six-week course just to be able to make toast with your toaster. Ponder if you dare the impact of similar complexity on the efficient operation of your electric toothbrush. Except for those holding the Certified Powered Dental Cleansing Appliance Operator designation, we'd all be toothless.

I didn't intend to draw a parallel between operating systems and dental hygiene when I started out, but that's the nature of creative writing. Sometimes when you dig for gold you come across earthworms instead. When that happens it's time to go fishing.

I'll be out in the boat.

Book Reviews

MARK LAMOURINE

The Essential Turing: The Ideas That Gave Birth to the Computer Age
B. Jack Copeland, ed.
Oxford University Press, 2004, 614 pages
ISBN 978-0-19-825080-7

It's really good to see Alan Turing finally getting his due in the popular media. He's been a large figure in the mathematical foundations of modern computing from the 1930s (along with John von Neumann and Emil Post, to name just a couple) for quite a long time. Despite this, and despite the fact that Turing's work is often glossed in elementary computing texts (who hasn't at least heard of a Turing machine?), the actual papers on which his reputation is based are not often studied by students of computer science or system administration. It's certainly not necessary anymore than it's necessary to read Copernicus, Galileo, or Newton in the original Latin to be able to do physics or calculus, or to read Euclid in Greek to do geometry. For me, though, something draws me to those original texts.

Turing's work contains much more than his wartime work on Enigma and the justifiably well-known "On Computable Numbers." During his life Turing worked on mathematical topics in artificial intelligence and even artificial life, anticipating the discovery of DNA by positing a computational underpinning to the origin and formation of biological structures. Copeland presents 16 publications on these four topics, ranging from peer-reviewed papers, to a letter from Turing and three others at Bletchley Park that was hand delivered to Winston Churchill to request additional resources for their code-breaking work, to personal mail to his mother during his stay at Princeton before the war. In each case, Copeland provides background and context to help the reader fully appreciate the main texts.

Many of the examples and arguments in Turing's essays may seem obscure or dated to someone who is already familiar with lambda calculus (through the use of Lisp or other modern functional programming languages). A number of them have a decidedly mathematical rather than computational bent, which is understandable when you realize that Turing was writing at a time when no real machines existed or were even under development. It remains remarkable to me that Turing and his colleagues, Alonzo Church, von Neumann, Post and others, conceived these ideas entirely in the abstract. When contrasted with today's methods of prototyping and fast-failure, the rigor involved is impressive (at least to a non-mathematician like myself).

The computational and mathematical writings here are presented in essence and more clearly in modern texts. If you are already familiar with Turing and his work through popular media or formal education in computer science and software development, you are unlikely to learn anything essential to your work. But you will gain insight into the range of topics to which Turing contributed and to the times and environment in which he worked as well as the pleasure of working through his original presentations.

I hope you will.

Postscript: If this kind of reading appeals to you, you might also be interested to find that Stephen Hawking has, over the past decade, released several edited volumes containing the foundational works of classical physics, quantum physics, and mathematics, translated into English and annotated, for those, like me, who feel the call to read them: *On the Shoulders of Giants: The Foundations of Physics and Astronomy; The Dreams That Stuff Is Made of: The Most Astounding Papers of Quantum Physics—and How They Shook the Scientific World;* and *God Created the Integers: The Mathematical Breakthroughs That Changed History.*

Drift into Failure
Sydney Dekker
Ashgate Publishing Ltd., 2011; 220 pages
ISBN 978-1-409402221-1

I think the most striking thing I found in *Drift into Failure* was the final section of the first chapter. That section was titled "Why we must not let Drift into Failure become the next folk model." (The previous section was titled "Great title, lousy metaphor.") The entire chapter was a sort of apology, although I think it was meant to set a framework for the rest of the book.

You see, the book is about how to think about failure, and more precisely, how to think about and analyze events in complex (nonlinear, to use a mathematical term) systems. The first and most important feature of these systems is that they will exhibit unpredictable behavior at times. This is the very nature of complexity, which brings us to the title of the first chapter as a whole (I'm working my way out of the Russian doll I built): "Failure Is Always an Option." If you're looking for a way to eliminate failure, you're reading the wrong book. Or, more significantly, you're doomed to fail, and you should understand how the world really works and pick an achievable goal: understanding how failures happen, looking for the human behaviors that increase the likelihood of failure (they're not what you would think), and knowing when *not* to waste money "solving" a problem that will never happen again.

Throughout the book, Dekker seems to be aware of the tug of human nature. This is explicit in what he writes but also seems to influence how he writes it. We, modern humans, both by nature (psychology) and culture (the legacy of Descartes and Newton), expect the world to work in a predictable, mechanical, linear way. Asking us to give up the certainty of the Clockwork Universe is a tough sell. We want to be safe, we want to be in control. The argument too is tough: "You want to be certain? You can't be certain, give up." We all answer, "Of course we can! Watch," and we find The Part That Failed and say, "There! That proves I can." Dekker is in the position of trying to prove a negative, to show that while you can always isolate "the cause of failure" after the fact, you cannot in principle prevent all failures by eliminating all points of failure. He knows this and is careful never to offer "the solution."

It's scary to realize that we are not in control *in the way we want to be*. Dekker's argument is that we have two choices: ignore the fact that we're building and depending on complex systems and continue to waste time and effort trying in vain to be 100% safe, or accept that failure is inevitable, but learn to minimize it systematically rather than reductively.

Dekker is trying to show that what we get through our reductionist impulses isn't what we think it is. In that quest he lays out a series of well-known catastrophic failures of technology and analyzes the analysis of the failure and response to the findings. These failures range from a single point mechanical failure that brings down an airliner to the systemic collapse of Enron. Each resulted at its root not from some point failure, but from a series of small, localized, apparently rational decisions that, when seen from a higher scale and in light of the now-apparent flaw, look reckless or even criminal. With each example, he comments on how the seed of a response that would have avoided the failure was already in place, but was minimized or ignored.

Dekker's conclusion is that we, as a society, must change. We must learn to accept risk and failure and respond not by punishing the whistleblowers and the outliers who raise flags before failures, but by encouraging them and listening to them. He advocates creating businesses and other social structures where variety and diversity are accepted, welcomed, and rewarded, because these produce resilient systems. This is a message that has been espoused and championed in the last decade in the software development and service industries as DevOps and Agile methods. More recently, more mainstream businesses have picked up the banner and are finding that, when well done and used appropriately, these methods can work.

There are also cases of both misuse and of failure even when these methods are applied appropriately. The whole point of the book is that failure is inevitable, but that risk is manageable. I think Dekker's reserved tone comes from his understanding of how human nature and modern media, with their two-sides-to-everything mentality, will misrepresent his ideas and lead to a misguided and doomed popular movement akin to the common pop culture abuse of the terms of evolution in places where it just doesn't apply.

For someone able to make a close and careful reading, Dekker will help create a framework with which to begin thinking and working to understand and (as much as is possible) control complex systems in work and in real life. I'm not sure he'll be able to convince the general public, as wedded as it is to a reductionist world model and as insistent on Keeping Me Safe and Finding Someone To Blame as it is. I can only hope.

USENIX ASSOCIATION FINANCIAL STATEMENTS FOR 2014

The following information is provided as the annual report of the USENIX Association's finances. The accompanying statements have been reviewed by Michelle Suski, CPA, in accordance with Statements on Standards for Accounting and Review Services issued by the American Institute of Certified Public Accountants. The 2014 financial statements were also audited by McSweeney & Associates, CPAs.

Accompanying the statements are charts that illustrate the breakdown of the following: operating expenses, program expenses, and general and administrative expenses. The operating expenses for the Association consist of the following: program expenses, management and general expenses, and fundraising expenses, as illustrated in Chart 1. The operating expenses include the general and administrative expenses allocated across the Association's activities. Chart 2 shows the breakdown of USENIX's general and administrative expenses. The program expenses, which are a subset of the operating expenses, consist of conferences and workshops; membership (including *;login:* magazine); projects, programs, and good works projects; their individual portions are illustrated in Chart 3.

The Association's complete financial statements for the fiscal year ended December 31, 2014, are available on request.

Casey Henderson, Executive Director

USENIX ASSOCIATION
STATEMENTS OF FINANCIAL POSITION
As of December 31, 2014 & 2013

ASSETS	2014	2013
Current Assets		
Cash & cash equivalents	$ 252,948	$ 217,555
Receivables	83,740	19,017
Prepaid expenses	91,704	53,434
Total current assets	428,392	290,006
Investments at fair market value	5,096,241	5,066,749
Property and Equipment		
Office furniture and equipment	379,872	370,609
Website	700,765	700,765
Leasehold improvements	32,217	29,631
Less: accumulated depreciation	(749,924)	(594,491)
Net property and equipment	362,930	506,514
Other assets	-	79,371
	$ 5,887,563	$ 5,942,640
LIABILITIES AND NET ASSETS		
Current Liabilities		
Accounts payable	$ 41,967	$ 109,667
Accrued expenses	57,203	42,466
Evi Nemeth Student Fund	1,525	1,250
Deferred revenue	91,080	31,540
Total current liabilities	191,775	184,923
Long-Term Liabilities	-	79,371
Total liabilities	191,775	264,294
Net Assets		
Unrestricted Net Assets	5,695,788	5,678,346
Net Assets	5,695,788	5,678,346
	$ 5,887,563	$ 5,942,640

USENIX ASSOCIATION
STATEMENTS OF ACTIVITIES
For the Years Ended December 31, 2014 & 2013

	2014	2013
REVENUES		
Conference & workshop revenue	$ 3,598,143	$ 2,860,442
Membership dues	281,847	300,666
Product sales	7,360	7,121
Event services & projects	111,088	
LISA SIG dues & other revenue	44,046	49,749
General sponsorship	85,500	25,000
Total revenues	4,127,984	3,242,978
OPERATING EXPENSES		
Conferences & workshops	3,268,065	2,891,344
Membership ;login:	240,194	456,418
Projects, programs & good works	208,395	121,838
LISA SIG expenses	3,586	71,001
Management and general	420,976	348,244
Fund raising	97,276	93,426
Total operating expenses	4,238,492	3,982,271
Net operating deficit	(110,508)	(739,293)
NON-OPERATING ACTIVITY		
Donations	23,197	23,305
Interest & dividend income	190,655	139,920
(Losses) gains on marketable securities	(26,339)	514,185
Investment fees	(59,895)	(60,097)
Other non-operating	332	42
Net investment income & non-operating expense	127,950	617,355
Change in net assets	17,442	(121,938)
Net assets, beginning of year	5,678,346	5,800,284
Net assets, end of year	$ 5,695,788	$ 5,678,346

Continued on page 80

Chart 1: USENIX 2014 Operating Expenses

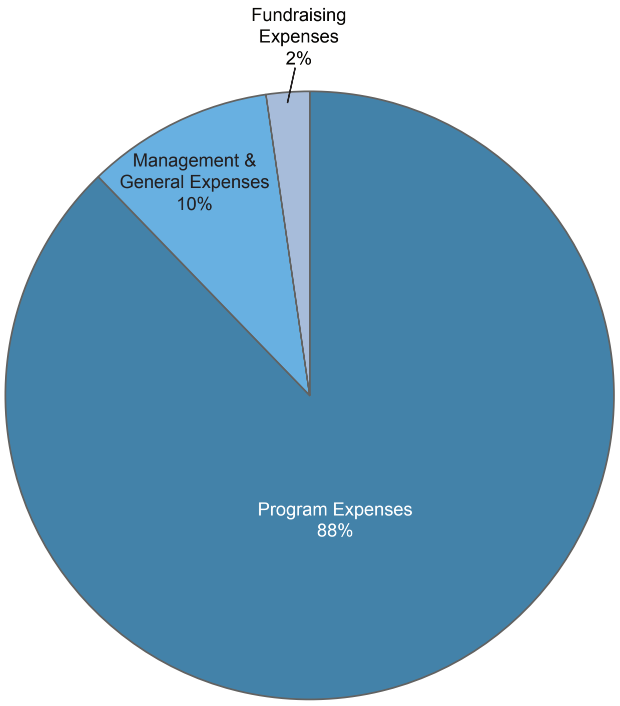


Chart 2: USENIX 2014 General & Administrative Expenses

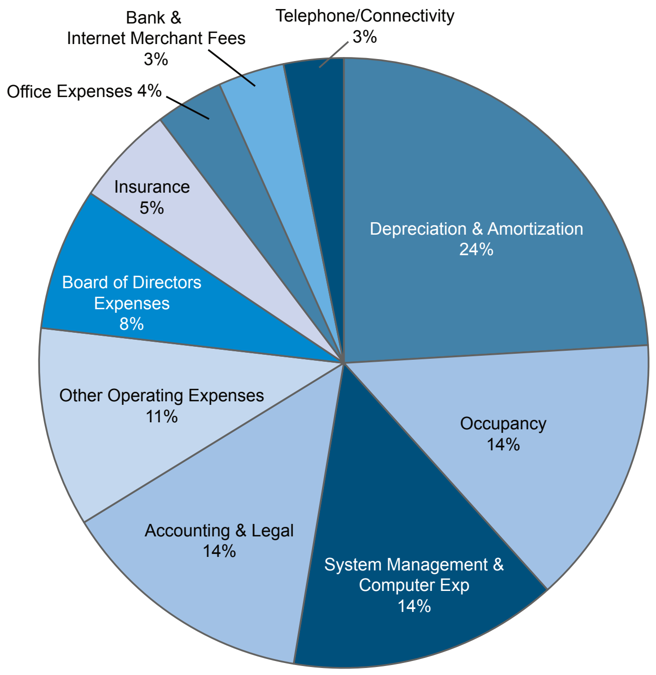
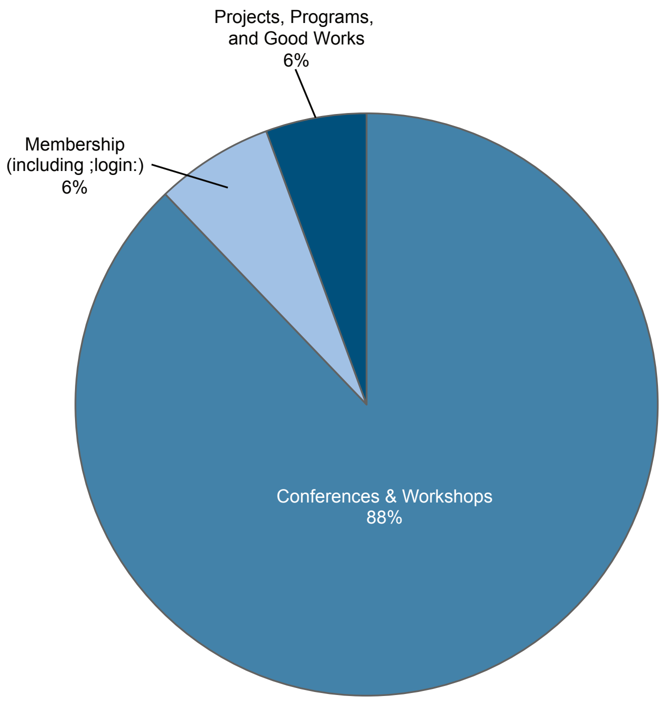


Chart 3: USENIX 2014 Program Expenses



USENIX
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

USENIX Association
2560 Ninth Street, Suite 215
Berkeley, CA 94710

POSTMASTER
Send Address Changes to *;login:*
2560 Ninth Street, Suite 215
Berkeley, CA 94710

PERIODICALS POSTAGE
PAID
AT BERKELEY, CALIFORNIA
AND ADDITIONAL OFFICES

usenix 40TH ANNIVERSARY

usenix
LISA15

More craft.
Less cruft.

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

LISA15 will feature talks and training from:

- Mikey Dickerson, U.S. Digital Service
- Nick Feamster, Princeton University
- Matt Harrison, Python/Data Science Trainer, Metasnake
- Elizabeth Joseph, Hewlett-Packard
- Tom Limoncelli, SRE, Stack Exchange, Inc
- Dinah McNutt, Google, Inc.
- James Mickens, Harvard University
- Chris Soghoian, American Civil Liberties Union
- John Willis, Docker

Nov. 8–13, 2015
Washington, D.C.

EARLY BIRD DISCOUNT
REGISTER BY OCTOBER 15, 2015

usenix.org/lisa15

Sponsored by USENIX in cooperation with LOPSA