SPRING 2019 VOL. 44, NO. 1

A POSIX Kernel Written in Go
Cody Cutler, M. Frans Kaashoek, and Robert Morris

Pocket for Container Storage
Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis

Noria: A Fast, In-Memory Database
Jon Gjengset, Malte Schwarzkopf, Jonathan Behrens, Lara Timbó Araújo, Martin Ek, Eddie Kohler, M. Frans Kaashoek, and Robert Morris

Boring Tech Best for SRE
Dave Mangot

Columns

Programming Puzzles and Trouble with Python's Zip
Peter Norton

The End of an Era
David N. Blank-Edelman

Executing Binaries from Go
Chris (Mac) McEniry

Monitoring Flow, Part 3
Dave Josephsen

Security Patents by Country and Type
Dan Geer and Scott Guthery

UPCOMING EVENTS

SREcon19 Americas
March 25–27, 2019, Brooklyn, NY, USA
www.usenix.org/srecon19americas

OpML '19: 2019 USENIX Conference on Operational Machine Learning
May 20, 2019, Santa Clara, CA, USA
www.usenix.org/opml19

SREcon19 Asia/Australia
June 12–14, 2019, Singapore
www.usenix.org/srecon19asia

2019 USENIX Annual Technical Conference
July 10–12, 2019, Renton, WA, USA
www.usenix.org/atc19

Co-located with USENIX ATC '19

HotStorage '19: 11th USENIX Workshop on Hot Topics in Storage and File Systems
July 8–9, 2019
www.usenix.org/hotstorage19

HotCloud '19: 11th USENIX Workshop on Hot Topics in Cloud Computing
July 8, 2019
www.usenix.org/hotcloud19

HotEdge '19: 2nd USENIX Workshop on Hot Topics in Edge Computing
July 9, 2019
www.usenix.org/hotedge19

SOUPS 2019: Fifteenth Symposium on Usable Privacy and Security
August 11–13, 2019, Santa Clara, CA, USA
Co-located with USENIX Security '19
Submissions for posters and lightning talks due May 24, 2019
www.usenix.org/soups2019

28th USENIX Security Symposium
August 14–16, 2019, Santa Clara, CA, USA
Co-located with SOUPS 2019
www.usenix.org/sec19

Co-located with USENIX Security '19

PEPR '19: 2019 USENIX Conference on Privacy Engineering Practice and Respect
August 12–13, 2019

WOOT '19: 13th USENIX Workshop on Offensive Technologies
August 12–13, 2019
Submissions due May 29, 2019
www.usenix.org/woot19

CSET '19: 12th USENIX Workshop on Cyber Security Experimentation and Test
August 12, 2019
Submissions due May 21, 2019
www.usenix.org/cset19

ScAINet '19: 2019 USENIX Security and AI Networking Conference
August 12, 2019
Talk submissions due March 28, 2019
www.usenix.org/scainet19

FOCI '19: 9th USENIX Workshop on Free and Open Communications on the Internet
August 13, 2019
Submissions due May 23, 2019
www.usenix.org/foci19

HotSec '19: 2019 USENIX Summit on Hot Topics in Security
August 13, 2019
Lightning talk submissions due June 10, 2019
www.usenix.org/hotsec19

SREcon19 Europe/Middle East/Africa
October 2–4, 2019, Dublin, Ireland

LISA19
October 28–30, 2019, Portland, OR, USA

Enigma 2020
January 27–29, 2020

FAST '20: 18th USENIX Conference on File and Storage Technologies
February 24–27, 2020
Co-located with NSDI '20
Submissions due September 26, 2019
www.usenix.org/fast20

NSDI '20: 17th USENIX Symposium on Networked Systems Design and Implementation
February 25–27, 2020, Santa Clara, CA, USA
Co-located with FAST '20
Fall paper titles and abstracts due September 12, 2019
www.usenix.org/nsdi20

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

EDITORIAL

2 **Musings** *Rik Farrow*

SYSTEMS

6 **The Benefits and Costs of Writing a POSIX Kernel in a High-Level Language** *Cody Cutler, M. Frans Kaashoek, and Robert Morris*

11 **Pocket: Elastic Ephemeral Storage for Serverless Analytics** *Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis*

17 **Noria: A New Take on Fast Web Application Backends** *Jon Gjengset, Malte Schwarzkopf, Jonathan Behrens, Lara Timbó Araújo, Martin Ek, Eddie Kohler, M. Frans Kaashoek, and Robert Morris*

SRE

22 **Achieving Reliability with Boring Technology** *Dave Mangot*

27 **Anticipating and Dealing with Operational Debt** *Laura Nolan*

PROGRAMMING

29 **How to Reinvent the Bicycle** *Sergey Babkin*

MACHINE LEARNING

35 **From Data Science to Production ML: Introducing USENIX OpML** *Nisha Talagala, Bharath Ramsundar, and Swaminathan Sundararaman*

COLUMNS

38 **Python** *Peter Norton*

42 **Practical Perl Tools: So Long and Thanks for All the Fish** *David N. Blank-Edelman*

44 **Executing Other Programs in Go** *Chris (Mac) McEniry*

49 **iVoyeur: Flow 3** *Dave Josephsen*

53 **For Good Measure: Patent Activity as a Measure of Cybersecurity Innovation** *Dan Geer and Scott Guthery*

56 **/dev/random: Ambush Computing** *Robert G. Ferrell*

BOOKS

58 **Book Reviews** *Rik Farrow and Mark Lamourine*

USENIX NOTES

61 **Notice of Annual Meeting**

61 **Community Survey: Some Answers, Some More Questions** *Liz Markel*

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 quarterly by the USENIX Association, 2560 Ninth Street, Suite 215, Berkeley, CA 94710.

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

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

©2019 USENIX Association
USENIX is a registered trademark of the USENIX Association. Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. USENIX acknowledges all trademarks herein. Where those designations appear in this publication and USENIX is aware of a trademark claim, the designations have been printed in caps or initial caps.

EDITORIAL

Musings

RIK FARROW

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

In 1983, I wrote one of my first attempts at fiction: a day in the life of a system administrator. My intent was to provide examples of what system administrators of UNIX systems typically did: boot the system, check logs, solve problems with printers or full disks, update software, and back up the system. These tasks didn't require a full-time position as there was just one system to manage. That was my vision at the time, and you can see my story at [1], scanned from the documentation I wrote.

When I started writing my sysadmin book, I did more research. My first sally into the field had been based on documentation I was asked to write for systems that were being sold to organizations that needed a way to have multiple people share that same system, using terminals. I visited UC Berkeley, where they had four VAXen running BSD, and talked to sysadmins there, who had no idea they were acting as sysadmins. They were students tasked with freeing up disk space, unclogging print queues, backing up, adding or deleting users, in other words, pretty much what I had imagined sysadmins would be doing. My vision of the world of sysadmin seemed well-aligned with reality.

For many years after that initial research in 1985, the world of the sysadmin remained mostly unchanged. Sysadmins were very often chosen from the ranks of Liberal Arts majors (although no college education was required) because they expressed an interest and willingness to work with UNIX systems or, perhaps, because they were drafted unwillingly to work on computers, especially anything as arcane as UNIX appeared to be. Not that UNIX had the corner on weirdness. Based on my own experience by that time, UNIX was refreshingly consistent and easy to use compared to the Microsoft or IBM systems of the day.

Things had begun to change by the late '80s, and that change was marked by the creation of the LISA conference. LISA, or Large Installation System Administration, was founded as a conference to help those who had so many systems to manage that the older methods of doing so, sitting at each console and typing away, were no longer sufficient. Over time, the need for automation led to infrastructure as code, beginning with tools to manage configurations like CFEngine and NIS [2].

Over the next decade, workstations multiplied like bacteria, covering desktops at most organizations. These fell into two classes: UNIX systems, because they supported networking, and PCs running Novell Netware, as Windows didn't support either file sharing or networking until 1996. Novell featured centralized administration, while those managing fleets of UNIX workstations had to get creative, and often did so by building their own set of tools. You could say that we had whole networks of pets (as opposed to cattle [3]) in the '90s, and each set of pets was ruled by idiosyncratic tools, largely unportable to other organizations.

Sea Change

What changed everything was yet another startup: Google. Larry Page and Sergey Brin wanted to create an efficient way to index the Internet. Companies had built indexes mostly manually up to that time, meaning that you searched through these indexes hoping they might lead you to the information or product you were really searching for. Page and Brin had

another idea and that was to actually canvas the web, collect all the pages they could find, then index and rank those pages based on the data the pages contained and the links that referred to those pages.

Besides needing serious network bandwidth, Page and Brin also needed a lot of computing horsepower, and that was seriously expensive. Their approach was to divide up the task so that a collection of computers could do the job, creating a form of parallelism that was fairly new at the time and terribly common today.

As Google grew, so did their clusters of computers. The service provided by those clusters also grew over time, so different clusters would be providing different services. But the clusters themselves were designed to be pretty interchangeable from the start. And managing those clusters of Linux systems had also moved very far away from being the system administrator of a bunch of desktops and a handful of servers. Google had invented cattle.

Ben Treynor Sloss, VP of Engineering at Google, invented the term Site Reliability Engineer (SRE). Ben had started out as a developer who got moved into operations in 2003, and decided to run his operations team as he would a developer team. Ben also came up with other important concepts, such as the error budget. That is, if your service-level objective (SLO) is 98.6%, that remaining 1.4% is your error budget: the amount of time your team has for updates or handling service outages.

There's much more to being an SRE, and one of the most important concepts, in my opinion, is eliminating toil. Toil is repetitive work that can be automated away, and SREs are supposed to spend no more than 50% of their time on operations so they can spend the rest of the working time on automation. As Sloss has mentioned, as you grow, your operations may scale exponentially. But your operations staff cannot scale exponentially. You must automate.

Even as SRE concepts became more popular, there has been a lot of pushback: not all organizations are going to be like Google, Facebook, and LinkedIn, to name a few. But what are most organizations today doing with their computing infrastructure? They are moving to the cloud, and if they expect to scale up their operations, they too will need to behave more like organizations with SREs.

The Lineup

We open this issue with three articles based on papers presented at OSDI '18. There were many more papers at OSDI of course [4], but I picked these because I liked them and thought they would be of broad interest to USENIX members.

Cody Cutler, Frans Kaashoek, and Robert Morris wrote an experimental operating system using Go. Their original goal had been to see whether language features would be an aid in OS writing, but the project pivoted toward seeing whether a high-level language, one with garbage collection, could run popular applications as fast as Linux could.

The next two articles include open source projects that support running services in clouds or clusters. Ana Klimovic et al. created Pocket as a means for ephemeral storage. Services that need to quickly store short-lived objects, such as Spark, are poorly served by the current mix of cloud storage, like S3. Pocket manages a range of storage services that are both faster and cheaper to use than current offerings.

Jon Gjengset et al. wrote Noria, a database server with an SQL front end that is not only faster than existing servers, like MySQL, but also supports much higher loads. Noria is a dataflow processing system that creates graphs where the vertices are operators and edges carry updates. Noria is slower to write but much faster for reads, and fits best when applications have read-heavy mixes.

While at LISA18, I heard several people talking about boring tech. That sounds, well, boring, but it's actually about keeping your architecture as simple as possible. I met Dave Mangot, who had presented "Familiar Smells" [5] and stirred up a fair bit of controversy. Dave agreed to explain his points about how important it is to architect your systems and services so that they are as simple as possible.

Laura Nolan volunteered to write a column about operational debt. You probably have heard of technical debt. Laura compares technical debt to credit card debt, but likens operational debt to a mortgage. Operational debt has to do with having failed to automate as much of operations as possible and instead having to waste most of one's time on toil.

Sergey Babkin offered to write about his experience interviewing people for mid-level programming positions. Sergey's thesis is that when it comes to solving the programming problems that often are used during interviews, he sees people using two different approaches. Each approach has its strengths, but they are best used together rather than in isolation.

Nisha Talagala, Bharath Ramsundar, and Swami Sundararaman wrote about the new, one-day OpML conference happening in May 2019. With the huge surge in interest in machine learning (ML), they discovered that just as there is a need for AI specialists, there's also a growing need for people who can run ML at scale. ML involves not just AI, but also working with vast amounts of data as well as other production issues.

Peter Norton explores issues with `zip`, a function used to create iterables in Python. Peter had been stretching his skills using the Advent of Code and, while solving one of the problems, uncovered a weakness in how `zip` works. His workaround, `SnitchZip`, is simple, but it won't be replacing `zip`.

David Blank-Edelman is retiring his column after having written it 66 times. At least that's what his final Perl program has discovered. We thank David for being so generous with his time over the last 12 years, and sharing his approaches to problem-solving with a Perl flair.

Mac McEniry shows us how to execute commands from within a Go program. Mac breaks down the potential usage into groups, depending on input and output, and whether to wait, forget, pipe-in, check out, or replace data.

Dave Josephsen expands on the monitoring system for mail processing at Sparkpost. In Part 3, Dave focuses on detecting backed-up queues within Fluentd, as well as staying with his theme on rivers and flow. With the sewers of Paris backing up, the flows aren't so fragrant.

Dan Geer and Scott Guthery examine the patterns of patents granted that relate to cybersecurity. Over time, the preponderance of new patents has shifted from the US and Europe to Asia, even as the general topics of security-related patents has changed over a period of decades.

Robert Ferrell discusses the reality of ubiquitous computing. With everything from online doorbells to toilets, Robert still manages to leave out automotive systems like OnStar that tell GM every time you accelerate or stop too quickly. But like the systems Robert describes, all of the data gathered is for the use of our corporate overlords.

Mark Lamourine has written three book reviews, covering managing Kubernetes, learning Git, and using "gamestorming." I reviewed *The Site Reliability Workbook,* the follow up to *Site Reliability Engineering*.

In USENIX Notes, I interview Liz Markel, the new Community Engagement Manager. You will be seeing Liz, often with her camera handy, at future USENIX events.

System administration has morphed from managing single servers to riding herd on fleets of cattle. While there will always be pets, especially in organizations that are naturally disposed to being fiefdoms, like many universities, the world has changed. To be honest, I think many people are glad that they don't have to design their own systems for fleet management and that the tooling has become so much more powerful over time.

References

[1] A Day in the Life of a System Administrator: https://rikfarrow.com/sysadminday.html.

[2] O. Kirch and T. Dawson, "The Network Information System," Chapter 13 in *Linux Network Administrator's Guide, 2nd Edition* (O'Reilly Media, 2000): https://www.oreilly.com/openbook/linag2/book/ch13.html.

[3] R. Bias, "The History of Pets vs Cattle and How to Use the Analogy Properly," September 29, 2016: http://cloudscaling.com/blog/cloud-computing/the-history-of-pets-vs-cattle/.

[4] OSDI '18 Technical Sessions: https://www.usenix.org/conference/osdi18/technical-sessions.

[5] D. Mangot, "Familiar Smells I've Detected in Your Systems Engineering Organization and How to Fix Them," LISA18: https://www.usenix.org/conference/lisa18/presentation/mangot.

28TH USENIX SECURITY SYMPOSIUM

SANTA CLARA, CA, USA

Co-located Workshops

WOOT '19

13th USENIX Workshop on Offensive Technologies
August 12–13, 2019
Submissions due May 29, 2019
www.usenix.org/woot19

WOOT '19 aims to present a broad picture of offense and its contributions, bringing together researchers and practitioners in all areas of computer security. Offensive security has changed from a hobby to an industry. No longer an exercise for isolated enthusiasts, offensive security is today a large-scale operation managed by organized, capitalized actors. Meanwhile, the landscape has shifted: software used by millions is built by startups less than a year old, delivered on mobile phones and surveilled by national signals intelligence agencies. In the field's infancy, offensive security research was conducted separately by industry, independent hackers, or in academia. Collaboration between these groups could be difficult. Since 2007, the USENIX Workshop on Offensive Technologies (WOOT) has aimed to bring those communities together.

CSET '19

12th USENIX Workshop on Cyber Security Experimentation and Test
August 12, 2019
Submissions due May 21, 2019
www.usenix.org/cset19

CSET '19 invites submissions on cyber security evaluation, experimentation, measurement, metrics, data, simulations, and testbeds. The science of cyber security poses significant challenges. For example, experiments must recreate relevant, realistic features in order to be meaningful, yet identifying those features and modeling them is very difficult. Repeatability and measurement accuracy are essential in any scientific experiment, yet hard to achieve in practice. Few security-relevant datasets are publicly available for research use and little is understood about what "good datasets" look like. Finally, cyber security experiments and performance evaluations carry significant risks if not properly contained and controlled, yet often require some degree of interaction with the larger world in order to be useful.

ScAINet '19

2019 USENIX Security and AI Networking Conference
August 12, 2019
Talk proposals due March 28, 2019
www.usenix.org/scainet19

ScAINet '19 will be a single track symposium of cutting edge and thought-inspiring talks covering a wide range of topics in ML/AI by and for security. The format will be similar to Enigma but with a focus on security and AI. Our goal is to clearly explain emerging challenges, threats, and defenses at the intersection of machine learning and cybersecurity, and to build a rich and vibrant community which brings academia and industry together under the same roof. We view diversity as a key enabler for this goal and actively work to ensure that the ScAINet community encourages and welcomes participation from all employment sectors, racial and ethnic backgrounds, nationalities, and genders.

FOCI '19

9th USENIX Workshop on Free and Open Communications on the Internet
August 13, 2019
Submissions due May 23, 2019
www.usenix.org/foci19

FOCI '19 will bring together researchers and practitioners from technology, law, and policy who are working on means to study, detect, or circumvent practices that inhibit free and open communications on the Internet.

HotSec '19

2019 USENIX Summit on Hot Topics in Security
August 13, 2019
Lightning talk submissions due Monday, June 10, 2019
www.usenix.org/hotsec19

HotSec '19 aims to bring together researchers across computer security disciplines to discuss the state of the art, with emphasis on future directions and emerging areas. HotSec is not your traditional security workshop! The day will consist of sessions of lightning talks on emerging work and positions in security, followed by discussion among attendees. Lightning talks are 5 MINUTES in duration—time limit strictly enforced with a gong! The format provides a way for lots of individuals to share ideas with others in a quick and more informal way, which will inspire breakout discussion for the remainder of the day.

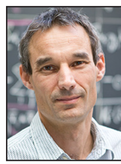
Registration will open in May 2019.

SYSTEMS

The Benefits and Costs of Writing a POSIX Kernel in a High-Level Language

CODY CUTLER, M. FRANS KAASHOEK, AND ROBERT MORRIS

Cody Cutler is a PhD candidate in computer science at MIT. Cody loves baffling bugs, performance optimization, and building systems. ccutler@csail.mit.edu



Frans Kaashoek is the Charles Piper Professor in MIT's EECS department and a member of CSAIL, where he co-leads the Parallel and Distributed Operating Systems Group (http://pdos.csail.mit.edu/). Frans is a member of the National Academy of Engineering and the American Academy of Arts and Sciences, and is the recipient of the ACM SIGOPS Mark Weiser award and the 2010 ACM Prize in Computing. He was a co-founder of Sightpath, Inc. and Mazu Networks, Inc. His current research focuses on multicore operating systems and certification of system software. kaashoek@mit.edu

Robert Morris is a Professor of Computer Science at MIT. rtm@csail.mit.edu

Biscuit is a POSIX-subset operating system kernel for x86_64 CPUs, which we wrote from scratch over the last four years. Biscuit is a bit more than a research toy. It can run Nginx and Redis with good performance and has some important operating system features, like multicore support, kernel-supported threads, a journaled file system, virtual memory, a TCP/IP stack, and device drivers for AHCI SATA disks and Intel 10 Gb network cards. Building Biscuit was a lot of fun and a lot of work.

Unlike most kernels, Biscuit is written in Go instead of C. C is the usual programming language choice for kernels because it can deliver high performance via flexible low-level access to memory and control over memory management (allocation and freeing). But C requires care and experience to use safely, and even then low-level bugs are common. For example, in 2017 at least 50 Linux kernel security vulnerabilities were reported that involved buffer overflow or use-after-free bugs in C code [7].

High-level languages (HLLs) have the potential to eliminate or reduce the impact of some common classes of bugs, particularly those having to do with memory and type safety. HLLs can also reduce programmer effort, thanks to automatic memory management, type safety, support for abstraction, and support for threads and synchronization.

However, OS designers have been skeptical about whether HLLs' memory management and abstraction are compatible with high-performance production kernels [9, 10]. Garbage collection (GC), runtime safety checks, and abstraction all cost CPU cycles, and many suspect that the benefits may not be worth the performance cost. For example, Rust [8] is partially motivated by the idea that GC cannot be made efficient; instead, the Rust compiler analyzes the program to partially automate freeing of memory.

Whether or not to use HLLs for kernels, then, requires an investigation of their performance in that context. There has been little research exploring this question, so we set out to shed a bit more light on it.

Our first step was to build a new POSIX-subset kernel, called Biscuit, in Go. Biscuit can run many programs that also run on Linux (after recompilation), so we were able to compare total application+kernel performance for Biscuit versus Linux. We did this for the Nginx and Redis servers, both of which make intensive use of the kernel. We found that throughput on Biscuit was within 10% of throughput on Linux, though this comparison should be taken with a grain of salt: although we examined both kernels' code and numerous CPU profiles to verify that they executed the applications' system calls in nearly the same way, we cannot completely rule out the possibility that Linux's performance was understated due to having many more features than Biscuit. Nevertheless, we suspect the performance difference between the two is approximately correct. To better focus on the HLL's impact on performance, we then measured the CPU overhead of Go's HLL features while running our applications on Biscuit. The CPU overhead of HLL features was at most 15%, with GC accounting for up to 3%. We presented these results in detail at the OSDI 2018 conference [11].

Paying a performance cost of 15% for the benefits of an HLL seems worthwhile in non-performance-critical situations. Similar tradeoffs regularly occur in existing kernels; for example, the Linux kernels included in Ubuntu and Debian have several compile-time features for security and debugging enabled. These features (hardened user copy, scheduling stats, and ftrace) reduce performance (by up to 25% in one microbenchmark), but most people probably don't disable them. Go has a performance cost, but it improves both security and programmability.

Readers may wonder why we used Go instead of Rust, given that Rust has no GC and thus wouldn't pay GC's performance price. We specifically wanted a language with GC in order to explore whether GC simplifies concurrent code.

In the remainder of the article, we will discuss a few challenges faced by HLL kernels, some benefits of HLL kernels, and reflect on our experience building Biscuit.

HLL Kernel Challenges

This section discusses some common concerns about HLLs and GC in kernels, and outlines what we learned about them while building Biscuit.

A kernel in Go cannot recover from low-memory situations since Go does not expose allocation failure. Linux and the BSDs handle kernel heap RAM exhaustion ("out of memory," or OOM) by returning NULL from the allocator; the calling kernel code must detect and handle the failure. Biscuit can't do this because Go implicitly allocates and does not have a way to indicate allocation failure.

Biscuit therefore uses a different approach: each kernel operation (system call, interrupt, etc.) reserves the maximum amount of heap RAM that the operation could possibly allocate before executing the operation. If the reservation isn't immediately available, the code waits until it is, after waking a separate thread that attempts to free heap memory by evicting from caches and perhaps by killing memory-hogging processes. The reservation guarantees that all allocations made by the operation cannot fail and thus no code is needed to detect and handle their failure. Additionally, since Biscuit waits for memory before executing the operation and thus while holding no locks, this approach cannot deadlock, a problem that Linux has struggled with [2, 3]. The challenging part is deciding how much memory each operation should reserve.

Fortunately, Go was helpful in overcoming this challenge: it turns out that it is easy to statically analyze Go code. We used publicly available static analysis packages to write a tool that inspects Biscuit's source and performs an analysis similar to escape analysis. The tool does most of the work of choosing reservation sizes, with reasonably tight bounds, but some manual effort is still required.

GC will use too much total CPU. The GC must follow the pointers in all live heap objects, which typically requires a RAM fetch per object. If there are millions of objects, the total time required can be on the order of hundreds of milliseconds. However, there are a couple of reasons why the CPU cycles used by the GC in practice is likely to be acceptably low.

Kernel heaps are typically small. Kernel heap objects are usually small metadata describing resources like files, sockets, virtual memory mappings, routing table entries, etc. The kernel heap does not contain large data items, such as user memory pages or file-cache pages. Few programs cause the kernel to accumulate millions of files, sockets, or noncontiguous virtual memory mappings. Thus the kernel heap typically uses a relatively small fraction of RAM even if user applications use many gigabytes of user memory.

To understand kernel heap sizes, we inspected four of MIT's big time-sharing machines. All four run Ubuntu Linux, had at least 79 users logged in, and had at least 800 processes with between 9 and 16 GB of total resident memory. The total kernel heap RAM (the sum of allocated and free kernel heap RAM) was less than 2 GB on each machine. On the OpenBSD desktop machine on which the first author edited this article, the total resident user memory is 1.4 GB, but the total kernel heap RAM is less than 170 MB.

One potential source of large kernel heaps is the vnode cache. Careful eviction of the vnodes may keep the number of kernel heap objects low without hurting application performance, depending on the access pattern.

If a large kernel heap is necessary, one can provision extra RAM to reduce the fraction of CPU time spent in GC. The collector only has to run when the kernel heap has no free space. Thus the amount of free heap RAM (and allocation rate) determines the frequency of GCs: doubling the amount of free heap RAM halves the frequency of GCs. Therefore, so long as a machine has enough extra RAM that can be donated to the kernel heap, the GCs can be made rare enough that total CPU cycles used by GC will be low.

We suspect that dedicating extra memory to kernel heaps will often be an acceptable cost: many applications probably wouldn't be affected if the RAM available to them or the buffer cache was decreased by a few hundred megabytes.

Finally, it may be possible to further reduce the CPU overhead even when there is little free heap RAM by modifying Go's GC to be generational. Generational collection is effective at reducing GC overhead for most programs, and we suspect Biscuit would benefit from it similarly.

GC pause times will be too long. Even if the interval between collections can be made long, the collector must eventually execute. If the collector causes kernel execution to pause for substantial periods, it could delay latency-sensitive tasks such as redrawing a moved mouse pointer or processing an urgent client request.

Go uses a technique called concurrent collection to reduce collection pauses. The main idea is to split the GC work into small units and interleave them with ordinary execution. The result is that individual pauses caused by GC will last only for the duration of a unit of work. There are still two potential problems. One is that smaller units of GC work are less efficient than larger ones. The other problem is that spreading collection work out over time increases the time that *write barriers* must be active. Write barriers are code the compiler inserts before each write that perform bookkeeping if a heap object is written during a collection. Concurrent collection therefore trades decreased pause times for decreased efficiency.

We measured the pauses caused by Biscuit's GC while running a kernel-intensive server, Nginx. The maximum single pause incurred by kernel GC was 115 microseconds. A given client request, however, may be delayed by multiple individual pauses. So we also measured the total accumulated pauses during each Nginx client request and found that the maximum was 582 microseconds. Such pauses are rare: less than 0.3% of Nginx requests spent more than 100 microseconds executing GC work.

Some applications can't tolerate even rare pauses of hundreds of microseconds, but we suspect that many can. For example, servers in one Google service had a 99th-percentile latency of 10 milliseconds [4].

The Go compiler will generate slower code than C compilers. Readily available C compilers have been optimized for decades. Go's compiler is comparatively young and must generate additional instructions for safety checks (bounds checks, nil-pointer checks, etc.) and write barriers.

We compared the performance of generated code from Go and GCC by modifying Biscuit and Linux to have near-identical code paths for two kernel-intensive microbenchmarks, pipe ping-pong, and zero-fill-on-demand page faults. We found that the Go versions were 15% and 5% slower than the C versions, respectively. The main reason pipe ping-pong is slower in Go is that it executes more instructions for safety checks and write barriers. The performance of the page fault handler in Go is closer to that of C because the generated instructions are less important: the main bottlenecks are the fundamental CPU operations of entering/exiting the kernel and copying the zero page.

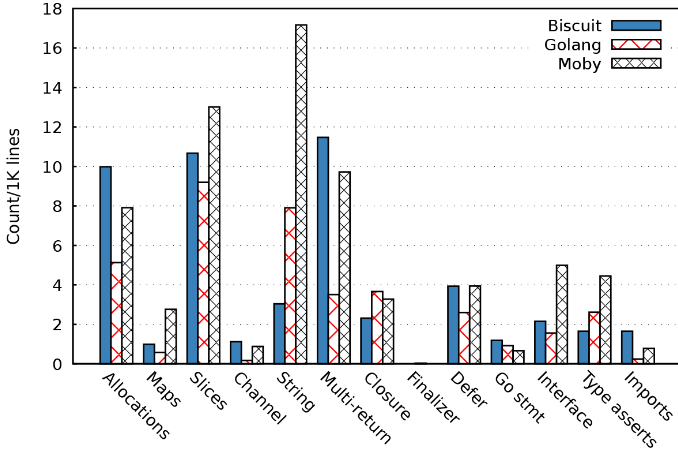


Figure 1: Uses of Go HLL features in the Git repositories for Biscuit, Golang, and Moby per 1,000 lines of code

Thus, for these two examples of typical kernel code, Go produced 5% to 15% slower executable code than C. For many situations, this is probably an acceptable price for the increased safety and programmability of Go.

HLL Kernel Benefits

Increased productivity. One of the main benefits of writing Biscuit in Go is the increased productivity over C. Unfortunately, we don't know a direct way of measuring productivity. Nevertheless, we believe Go significantly reduced the effort required to build Biscuit. Some of our favorite language features are GC'ed allocation, slices, defer, multi-value returns, closures, strings, and maps. Individually, none of these features are transformative, but together they result in significantly simpler code.

HLL features can increase productivity, but we weren't sure whether a kernel would be able to make good use of them. We compared the rate of use of several HLL features in Biscuit to two other large Go projects, Moby (https://github.com/moby/moby) and Golang (containing Go's compiler, runtime, and standard packages). Each bar in Figure 1 shows the number of uses of a particular feature per thousands of lines of code in the indicated project. Biscuit's use of most of the HLL features is in line with the other projects.

Memory safety. Manual memory management in C is error-prone, and the consequences of bugs can be severe: 40 out of the 65 publicly available, execute-code CVEs found in Linux during 2017 were due to manual memory management bugs, and all of them allow an attacker to execute malicious code in the kernel. Had this buggy code been written in Biscuit, the GC and runtime safety checks would have prevented malicious code execution in all 40 cases.

```
func serve() {
    buf := new(request_t)
    read_next_request(buf)
    go func() {
        // log_request() occasionally
        // blocks on IO
        log_request(buf)
    }()
    process_request(buf)
}
```

Listing 1: A simple case where threads share data

Simpler concurrency. Garbage collection makes threaded sharing of transient heap objects particularly convenient. For example, consider the request processing code in Listing 1. A network server calls the `serve` function to receive and process the next request. The code calls `log_request` in a separate thread in order to prevent file writes from delaying the processing of the request. Each thread accesses `buf` while logging or processing. The GC automatically ensures that `buf` will be freed only after both threads have finished using it.

In contrast, this style of threaded programming can be awkward in C, because of the need for code that decides when the last thread has finished using the object. Consider writing Listing 1 in C. The C programmer would allocate `buf` via `malloc`. Neither thread could simply free `buf` before returning since the other thread may still be accessing `buf`. The programmer must delay the call to `free` until both threads have finished accessing `buf`. One solution would be to embed a reference count in `buf`, manipulated with atomic instructions. This is eminently possible in C but requires more programmer thought than in Go, and thus more chance of error.

Simpler lock-free sharing. GC is convenient in the above example, but GC is more than convenient when threads share data without locks (which is common in optimized kernels [5]) because the resulting code is significantly simpler than in C. In C, each thread must increase and decrease the corresponding reference count before and after accessing an object. Forgetting to increase or decrease a reference count will result in corrupted or leaked memory. Since threads may concurrently modify the same reference counter, all modifications must be atomic with respect to other counter accesses. Furthermore, the reference counters themselves cannot be stored in the same memory as the object that they protect, since then a thread may modify freed memory. Thus the programmer needs to find the counter belonging to each object.

The atomic operations to maintain reference counts can reduce performance. This is the main reason why Linux uses RCU [5, 6] to safely free memory shared among threads. RCU requires significantly fewer atomic operations and thus achieves good performance, but it is not simple to use: code which accesses memory managed by RCU must follow a list of rules (see https://www.kernel.org/doc/Documentation/RCU/checklist.txt) and be surrounded by a special prologue and epilogue. All such code cannot sleep, schedule, or block in any way, in addition to following a few other rules.

GC makes these programming difficulties disappear. Biscuit code can share heap objects among threads without worrying about when to free the objects. The reduction of programmer effort is especially evident in the case of read-lock-free data structures, which Biscuit uses in its directory cache, routing table, and network interface table. The result is high performance with less programmer effort, particularly in the directory cache.

Experience and Reflections

Biscuit was a really exciting project because we had no idea what to expect of Go. Would Go make optimizing low-level code difficult or impossible? Can interrupt handlers tolerate GC pauses? Is a language runtime with its own state and invariants compatible with the degree of concurrency kernels have to handle? When we started, we expected to spend at most a couple of months on the project and quickly find an indisputable, concrete reason why a fast kernel could not be built in Go. We did not expect to end up with a kernel that runs Nginx and Redis on 10 Gb NICs with performance similar to Linux.

The focus of the project wasn't always performance. At the beginning, we hoped that Go's good support for threads and interthread communication and synchronization would allow simpler or more powerful designs for kernel code. For example, we hoped that a kernel in Go could make free use of transient worker threads to parallelize operations on multicore hardware. Unfortunately, we found few such situations. As a result, we switched goals away from exploring new kernel architectures and towards evaluating the effect of language choice and GC on performance. Thus the design of Biscuit started to become more and more traditional and similar to Linux in order to isolate performance differences due to the language as opposed to differing architectures.

Building an operating system is a huge amount of work, and it took months before Biscuit could run even the most trivial of programs. Biscuit currently has 58 system calls, and nearly all of them are required to run Nginx, Redis, and CMailbench.

As much work as it took to allow Biscuit to run complex programs, the optimization effort to run the programs well was far greater. We knew that Linux delivered good performance when we started, but we were stunned at how much effort it took to build a kernel whose performance was even within a factor of two of Linux's. Getting decent performance required implementing some interesting optimizations: mapping kernel text with

large pages to reduce iTLB misses, implementing TCP timers via streamlined timer-wheels, building a directory cache with store-free lookups that is correct with racing eviction, etc. But most were less interesting details: reducing lock contention by dedicating a NIC TX queue to each CPU instead of sharing one queue among all CPUs, avoiding unnecessary allocations or function calls, carefully batching TCP ACKs, sometimes using a linked list instead of an array, etc. Despite the effort, optimizing Biscuit's performance was the most fun part of the project and that's mainly because it honed our performance debugging skills. If we had to do it over again, we would write the code to profile via the CPU performance-monitoring counters as early as possible; those profiles were by far the most helpful tool for debugging performance problems.

We are grateful for QEMU [1], which has been a critical tool for building and testing Biscuit. We were amazed at how little work it took to get Biscuit to successfully boot on real hardware despite running it exclusively on QEMU up to that point. Real hardware did expose a few bugs in Biscuit (E820 memory map parsing, PCI interrupt routing, and the BIOS's INT 13h implementation apparently doesn't restore the interrupt flag), but it was generally painless, and that speaks to the quality of QEMU's emulation.

Our overall experience has been that building a kernel in Go was similar to building one in C: good kernel performance is more about implementing the right optimizations and less about the choice of programming language. Go didn't prevent us from implementing important kernel optimizations, which suggests that Go is a good choice for kernel programming.

Conclusion

Our experience using Go to implement the Biscuit kernel has been positive. Go's high-level language features are helpful in the context of a kernel. Examination of historical Linux kernel bugs due to C suggests that a type- and memory-safe language such as Go might avoid real-world bugs or handle them more cleanly than C. The ability to statically analyze Go helped us implement defenses against kernel heap exhaustion, a traditionally difficult task.

We measured some of the performance costs of Biscuit's use of Go's HLL features on a set of kernel-intensive benchmarks. The fraction of CPU time consumed by garbage collection and safety checks is less than 15%. We compared the performance of equivalent kernel code paths written in C and Go, finding that the C version is about 15% faster.

The paper and Biscuit's code are available at https://pdos.csail.mit.edu/projects/biscuit.html.

References

[1] QEMU, the FAST! processor emulator, 2018: https://www.qemu.org.

[2] J. Corbet, "The Too Small to Fail Memory-Allocation Rule," LWN.net, December 2014: https://lwn.net/Articles/627419/.

[3] J. Corbet, "Revisiting Too Small to Fail," LWN.net, May 2017: https://lwn.net/Articles/723317/.

[4] J. Dean and L. A. Barroso, "The Tail at Scale," *Communications of the ACM*, vol. 56, no. 2, February 2013, pp. 74–80.

[5] P. E. McKenney, S. Boyd-Wickizer, and J. Walpole, "RCU Usage in the Linux Kernel: One Decade Later," 2012.

[6] P. E. McKenney and J. D. Slingwine, "Read-Copy Update: Using Execution History to Solve Concurrency Problems," in *Parallel and Distributed Computing and Systems*, 1998, pp. 509–518.

[7] MITRE Corporation, CVE Linux Kernel Vulnerability Statistics, 2018: http://www.cvedetails.com/product/47/Linux-Linux-Kernel.html?vendor id=33.

[8] S. Klabnik and C. Nichols, *The Rust Programming Language* (No Starch, 2018): https://doc.rust-lang.org/book/.

[9] A. S. Tanenbaum, *Modern Operating Systems* (Pearson Prentice Hall, 2008), p. 71.

[10] L. Torvalds, On C++, January 2004: http://harmful.cat-v.org/software/c++/linus.

[11] C. Cutler, M. F. Kaashoek, R. T. Morris, "The Benefits and Costs of Writing a POSIX Kernel in a High-Level Language," in *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI '18),* pp. 89–105: https://www.usenix.org/system/files/osdi18-cutler.pdf.

Pocket
Elastic Ephemeral Storage for Serverless Analytics

ANA KLIMOVIC, YAWEN WANG, PATRICK STUEDI, ANIMESH TRIVEDI, JONAS PFEFFERLE, AND CHRISTOS KOZYRAKIS

Ana Klimovic is a PhD student at Stanford University advised by Professor Christos Kozyrakis. Her research interests are in computer systems and computer architecture. She is particularly interested in improving performance and resource efficiency for cloud computing. Ana is a Microsoft Research PhD Fellow, Stanford Graduate Fellow, and Accel Innovation Scholar. anakli@stanford.edu

Yawen Wang is a second-year PhD student advised by Professor Christos Kozyrakis at Stanford University. She is broadly interested in computer systems and cloud computing. Her current research focuses on leveraging machine learning to manage cloud resources more efficiently. yawenw@stanford.edu

Patrick Stuedi is a researcher at IBM Research Zurich. His research interests are in distributed systems, networking, and operating systems. Patrick graduated with a PhD from ETH Zurich in 2008 and spent two years (2008–10) as a postdoc at Microsoft Research Silicon Valley. His work explores how modern networking and storage hardware can be exploited in distributed systems. Patrick is the creator of several open source projects such as DiSNI (RDMA for Java) and DaRPC (low latency RPC) and is co-founder of Apache Crail (Incubating). stu@zurich.ibm.com

Serverless computing platforms are increasingly being used to exploit massive parallelism and fine-grained billing for interactive analytics jobs [1–3]. A key challenge is exchanging intermediate data efficiently between tasks, as serverless tasks are short-lived and stateless. The systems commonly used to store and exchange intermediate data in serverless jobs today do not meet the performance, cost, and elasticity requirements of interactive analytics applications. We present *Pocket*, a fast, elastic, fully managed cloud storage service designed for efficient data sharing in serverless analytics applications. To achieve high performance and cost efficiency, Pocket leverages multiple storage technologies, right sizes resource allocations for jobs, and automatically scales cluster resources based on utilization. The system achieves similar performance to Redis, an in-memory datastore, while offering automatic, fine-grained scaling and significantly lower cost for serverless analytics jobs. Pocket is open source software [4].

Serverless Analytics

Serverless platforms like AWS Lambda, Google Cloud Functions, and Azure Functions enable users to quickly launch thousands of lightweight tasks, as opposed to entire virtual machines. Cloud providers automatically scale the number of serverless tasks based on application demands, and users pay only for the resources their tasks consume, at sub-second time granularity.

Though serverless computing was initially used for web microservices and IoT applications, its high elasticity and fine-grain billing make serverless computing appealing for more complex jobs, such as interactive analytics [1–3]. Analytics jobs typically consist of multiple stages of execution and require tasks in different stages to exchange data. We refer to the intermediate data shared between tasks as *ephemeral* (i.e., short-lived) data. We distinguish ephemeral data from the initial input and final output data of analytics jobs, which typically have longer lifetimes.

Traditional analytics frameworks (e.g., Spark) implement ephemeral data sharing with long-running framework agents buffering data in local storage on each node. In contrast, tasks in serverless deployments are short-lived, and any data that a task stores locally is lost when a task exits. Thus, direct communication between tasks is difficult, and the natural approach to share data is to use remote storage.

For instance, serverless analytics frameworks use object stores (e.g., S3), databases (e.g., CouchDB), or distributed caches (e.g., Redis).

Animesh Trivedi is a researcher at IBM Research Zurich. His interests are in anything and everything related to performance, ranging from multi-core CPUs to distributed environments. Currently, he is investigating how modern high-performance network and storage devices can be leveraged in large-scale data-processing systems such as Spark, Tensorflow, and serverless workloads. He is one of the founding members of the Apache Crail (Incubating) project. atr@zurich.ibm.com

Jonas Pfefferle is a Software Engineer at IBM Research Zurich in the Cloud Storage and Analytics group. His research interests are in virtualized distributed systems and datacenters, specifically in state-of-the-art network and storage technologies. Currently, he is working on exploiting high performance I/O devices with the focus on remote direct memory access (RDMA) and non-volatile memory (NVM) in data processing frameworks like Spark. Jonas holds a master's degree in computer science from ETH Zurich (2014). jpf@zurich.ibm.com

Christos Kozyrakis is a Professor in the Departments of Electrical Engineering and Computer Science at Stanford University. His research interests include resource-efficient cloud computing, energy-efficient computing and memory systems for emerging workloads, and scalable operating systems. Kozyrakis has a PhD in computer science from the University of California, Berkeley. He is a Fellow of the IEEE and ACM. kozyraki@stanford.edu

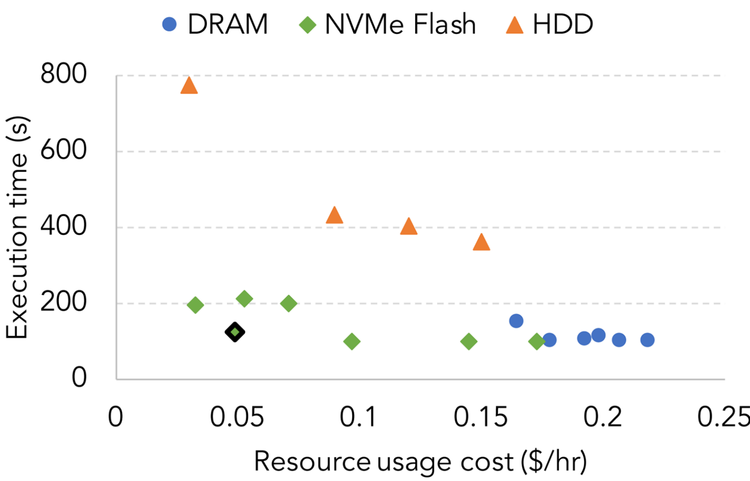


Figure 1: Example of the performance-cost tradeoff space for a serverless video analytics job using different storage technologies and VM types in Amazon EC2 for the ephemeral storage cluster. Data points of the same storage type represent applications using different numbers of nodes, compute resources, and network bandwidth.

However, existing storage services are not a good fit for sharing ephemeral data in serverless applications [5]. Popular fully managed cloud storage services, such as Amazon S3, are designed to store data with high durability and are not optimized for low latency or high elasticity. Distributed key-value stores offer good performance but burden users with managing the configuration and scale of a storage cluster. Selecting storage resource configurations for jobs is difficult yet critical for performance and cost efficiency [6]. Figure 1 shows an example of the performance-cost tradeoff for a serverless video analytics application using an ephemeral storage cluster configured with different storage technologies (DRAM, Flash, and disk), number of nodes, compute resources per node, and network bandwidth. Finding the minimum cost storage cluster configuration that provides optimal performance (e.g., the bold point in Figure 1) is nontrivial and gets even more difficult with multiple jobs.

Ephemeral Storage Requirements for Serverless Analytics

We study the ephemeral storage requirements of three different types of serverless analytics applications: distributed software compilation, video object recognition, and MapReduce sort. Figure 2 shows that ephemeral object size varies from 100s of bytes to 100s of megabytes. Hence, serverless analytics applications require both low latency, which is important for small object accesses, and high throughput, which is important for large object accesses. As serverless computing platforms elastically scale the number of tasks based on load, the ephemeral datastore must also be able to scale up and down automatically to meet dynamic I/O requirements while minimizing cost. In addition to rightsizing storage cluster resources based on current load, the system must place data on the right type of storage technology for each job by taking into account the latency, throughput, and cost tradeoffs of different technologies.

On the other hand, fault tolerance is not a high requirement for the ephemeral datastore as the data is short-lived, and application frameworks typically have built-in mechanisms, such as lineage tracking, that can be used to regenerate ephemeral data. Figure 3 shows the object lifetime CDF for the three serverless analytics jobs we study. Most ephemeral data objects only need to be stored for less than 30 seconds.

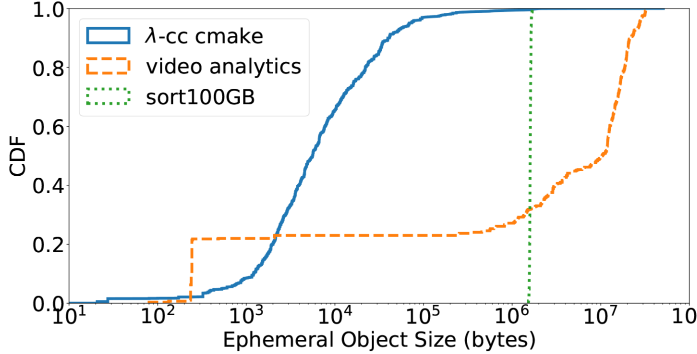


Figure 2: Ephemeral data object size CDF for three different serverless analytics applications. Objects vary widely in size.

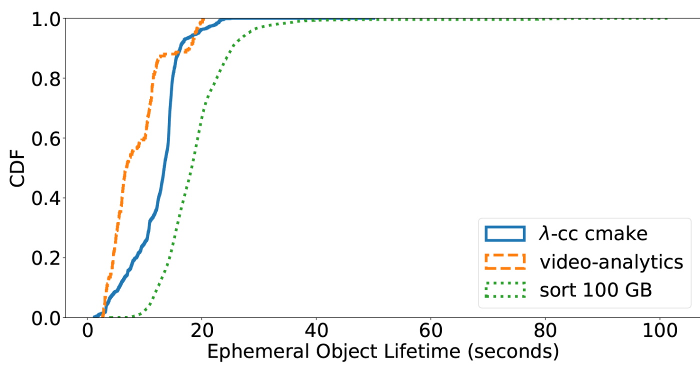


Figure 3: Ephemeral data objects have short lifetimes (seconds to minutes).

Ephemeral Storage as a Service

We introduce *Pocket*, an elastic storage service for ephemeral data sharing. The system provides high I/O performance while minimizing cost by leveraging multiple storage technologies with different performance-cost tradeoffs, rightsizing resource allocations for jobs, and automatically scaling cluster resources based on utilization. Pocket is a distributed `/tmp` for the cloud.

Pocket splits responsibilities across three separate planes: a control plane that determines data placement policies for jobs, a metadata plane that manages distributed data placement, and a metadata-oblivious data plane responsible for storing data. Pocket scales all three planes independently at fine granularity based on the current load. The system leverages optional hints about job characteristics, which can be provided by application frameworks or users via Pocket's API, to allocate the right storage technology, capacity, bandwidth, and CPU resources for each job. We intend for cloud operators to run Pocket as a fully managed storage service and charge users for only the storage capacity and bandwidth that their tasks consume.

Figure 4 shows Pocket's system architecture and how a job interacts with Pocket. To use Pocket, a job starts by registering with a logically centralized controller, which runs the control plane logic for Pocket. The controller decides the storage throughput, capacity, and type of storage technology to allocate for the job, leveraging any optional hints provided about the job's characteristics, such as latency sensitivity and the peak number of concurrent tasks. The controller decides on a data placement policy for the job, spinning up additional storage or metadata nodes if necessary to meet the job's allocation. The controller communicates the data-placement policy for the job to metadata servers, which will enforce data placement by routing client write requests. After registering with the controller, the job launches serverless tasks (i.e., lambdas), which issue GET/PUT requests using Pocket's client library.

Metadata servers route I/O requests to the appropriate storage nodes based on the job's data-placement policy determined upfront by the controller during job registration. By default, a job's ephemeral data is deleted when the job deregisters with the controller. However, Pocket's API accepts hints to manage data lifetime. For example, since we find that most ephemeral data is written and read only once, a user or application framework can hint to Pocket that an object should be deleted immediately after it is read, optimizing garbage collection.

In addition to rightsizing resource allocations across multiple dimensions upfront when jobs register, the controller also continuously monitors resource utilization in the cluster. Pocket's controller adds/removes nodes to keep CPU, storage capacity, and network bandwidth utilization within a target range. To balance load in a cluster of changing size, Pocket leverages the short-lived nature of ephemeral data and serverless jobs. We find that ephemeral data objects in the serverless applications we study typically only need to be stored for less than 30 seconds. Hence, migrating this data to redistribute load when nodes are added or removed would have high overhead. Instead, Pocket focuses on steering data for incoming jobs across active and new storage nodes in the cluster, while allowing nodes that the controller wants to take down to be drained as their data is garbage collected.

Implementation. Pocket's implementation leverages several open-source systems, and Pocket is also open source [4]. The metadata and data planes are built on top of the Apache Crail distributed datastore, which is designed for low latency, high throughput access to data with low durability requirements [7, 8]. Though Crail is originally designed to leverage RDMA networks, the system's modular architecture supports pluggable RPC libraries and storage tiers. We implement a TCP-based RPC library for Pocket. Our implementation of Pocket includes three different storage tiers. The first is a DRAM tier that efficiently serves client requests over TCP connections. The second tier is an NVMe Flash storage tier. We implement this tier using

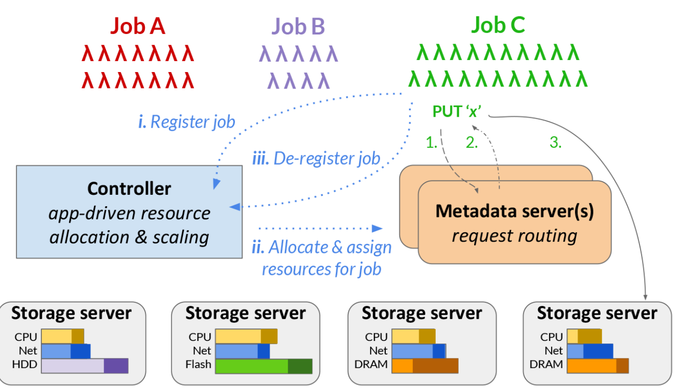


Figure 4: Pocket system architecture and the steps to register job C, issue a PUT from a lambda, and deregister the job. The shaded/colored bars on storage servers show used and allocated resources for all jobs in the cluster.

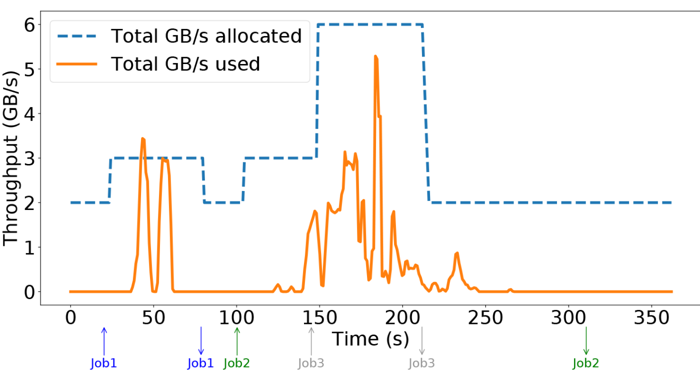


Figure 5: Pocket dynamically scales cluster resources to meet I/O requirements as jobs come and go.

ReFlex, a software system that enables fast, predictable access to remote Flash storage over commodity Ethernet networks [9]. The third tier is a generic block storage tier that allows Pocket to use any block device such as a hard-drive disk or SATA/SAS SSD with a standard kernel block device driver. We deploy Pocket storage and metadata servers inside of containers on AWS EC2 machines. We use Kubernetes to orchestrate the containers and implement autoscaling.

Elastic and Automatic Resource Scaling with Pocket

We evaluate Pocket with three different serverless analytics applications: a video analytics application that does object recognition, a MapReduce sort job, and a distributed compilation job that compiles the source code for `cmake`. The applications differ in their degree of parallelism, ephemeral object size distribution, and throughput requirements. We use AWS Lambda as our serverless computing platform.

Figure 5 shows how Pocket elastically scales cluster resources as multiple jobs register and deregister with the controller. In this experiment, we assume Pocket receives hints about the capacity and throughput requirements of each job. The first job is a 10 GB sort requesting 3 GB/s throughput. The second job does video object recognition, requesting 2.5 GB/s, and the third job is a different invocation of a 10 GB sort also requesting 3 GB/s. Pocket quickly and automatically scales the allocated storage bandwidth (dotted line) to meet application throughput demands (solid line). Application throughput briefly surpasses the total allocated throughput due to bursty EC2 VM network bandwidth, which causes a storage node to provide greater than the anticipated 1 GB/s bandwidth per node for a short period of time. In this experiment, the controller is configured to maintain a minimum cluster size of two storage nodes, which provides 2 GB/s cumulative throughput.

Comparing Pocket to Amazon S3 and Redis

We compare Pocket to two popular storage systems used by serverless analytics applications today. Amazon S3 is a fully managed cloud storage service that offers a convenient "serverless" storage abstraction and cost model in which users pay only for the capacity and bandwidth their tasks consume. S3 offers durable storage and is optimized for access to large objects. In contrast, Redis is a popular key-value store that uses DRAM to provide high performance. However, users need to manually select and manage resource configurations for a Redis storage cluster. Although Amazon and Azure offer managed Redis clusters through their ElastiCache and Redis Cache services, respectively, they do not automate storage management as desired by serverless applications. Users must still select instance types with the appropriate memory and compute and network resources to match their application requirements.

We first compare job execution time when using Pocket versus S3 and ElastiCache Redis as the ephemeral datastore. Figure 6 plots the per-task execution time breakdown for a 100 GB MapReduce sort job, run with 250, 500, and 1000 concurrent lambdas. The light-gray/orange bars show the time spent fetching original input data and writing final output data to long-term S3 storage, while the darker-gray/blue bars compare the time for ephemeral data I/O, comparing S3, Redis, and Pocket. S3 does not provide sufficient throughput for this I/O-intensive job, hence in the 250 lambda experiment, each task spends over three times longer shuffling data when using S3 compared to Redis or Pocket. When the job is run with 500 or more lambdas, S3 does not support sufficient request rates. The system returns errors and advises to reduce the I/O rate. On the other hand, Pocket provides similar throughput to Redis. In this experiment, we assume Pocket receives a hint that the job is not sensitive to latency—hence, Pocket uses NVMe Flash instead of DRAM. Thus Pocket achieves similar performance to Redis while dramatically saving cost.

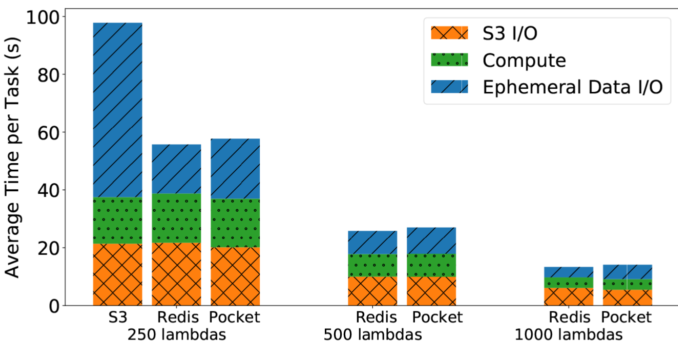


Figure 6: Average execution-time breakdown of each task (lambda) in a 100 GB sort job, run with 250, 500, and 1000 concurrent tasks

To compare the cost of running jobs using Pocket versus S3 and ElastiCache Redis for ephemeral data sharing, we derive a fine-grain resource cost model for Pocket based on Amazon EC2 pricing. Our minimum-size Pocket cluster, consisting of one controller node, one metadata server, and two NVMe Flash storage nodes, costs $1.63 per hour to run on Amazon EC2. However, Pocket's fixed cost can be amortized because the system is designed to support multiple concurrent jobs from one or more tenants. We intend for Pocket to be operated by a cloud provider and offered as a storage service with a pay-what-you-use cost model for users, similar to the cost model of serverless computing platforms. Hence, for our cost analysis, we derive fine-grain resource costs, such as the cost of a CPU core and the cost of each storage technology per GB, based on the prices of various EC2 instances.

Using this fine-grain resource pricing model for Pocket, we compare the cost of running the 100 GB sort, video analytics, and distributed compilation jobs with S3, ElastiCache Redis, and Pocket. For S3, we assume its GB-month cost is charged hourly. We base Redis costs on the price of entire VMs, not only the resources consumed, since ElastiCache Redis clusters are managed by individual users rather than cloud providers. Pocket achieves the same performance as Redis for all three jobs while saving 59% in cost. S3 is still orders of magnitude cheaper. However, S3's cloud provider-based cost is not a fair comparison to the cloud user-based cost model we use for Pocket and Redis. Furthermore, while the distributed compilation job has similar performance with all three ephemeral storage systems because it saturates CPU resources on serverless tasks, the execution time is 40 to 65% higher with S3 compared to Pocket for the video analytics and MapReduce sort jobs. A more detailed analysis of Pocket's performance and cost can be found in our OSDI '18 paper [10].

Conclusion

General-purpose analytics on serverless infrastructure presents unique opportunities and challenges for performance, elasticity, and resource efficiency. We analyzed the challenges associated with efficient data sharing and presented Pocket, a fully managed ephemeral data storage service. Pocket provides a highly elastic, cost-effective, and high performance storage solution for analytics workloads. Pocket achieves these goals using a strict separation of responsibilities for control, metadata, and data management. Although we designed Pocket specifically to enable efficient data sharing in serverless analytics applications, more generally, Pocket is a distributed temporary datastore that can be useful for a variety of different cloud applications.

Acknowledgments

We thank our OSDI shepherd, Hakim Weatherspoon, and the anonymous reviewers for their helpful feedback. We thank Qian Li, Francisco Romero, Sadjad Fouladi, and Nick Murphy for insightful technical discussions. This work is supported by the Stanford Platform Lab, Samsung, and Huawei. Ana Klimovic is supported by a Stanford Graduate Fellowship. Yawen Wang is supported by a Stanford Electrical Engineering Department Fellowship.

References

[1] S. Fouladi, R. S. Wahby, B. Shacklett, K. V. Balasubramaniam, W. Zeng, R. Bhalerao, A. Sivaraman, G. Porter, and K. Winstein, "Encoding, Fast and Slow: Low-Latency Video Processing Using Thousands of Tiny Threads," in *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI '17)*, pp. 363–376: https://www.usenix.org/system/files/conference/nsdi17/nsdi17-fouladi.pdf.

[2] E. Jonas, Q. Pu, S. Venkataraman, I. Stoica, and B. Recht, "Occupy the Cloud: Distributed Computing for the 99%," in *Proceedings of the Symposium on Cloud Computing (SOCC '17)*, pp. 445–451.

[3] Databricks Serverless, "Next Generation Resource Management for Apache Spark": https://databricks.com/blog/2017/06/07/databricks-serverless-next-generation-resource-management-for-apache-spark.html, 2017.

[4] Pocket: https://github.com/stanford-mast/pocket, 2018.

[5] A. Klimovic, Y. Wang, C. Kozyrakis, P. Stuedi, J. Pfefferle, and A. Trivedi, "Understanding Ephemeral Storage for Serverless Analytics," in *Proceedings of the USENIX Annual Technical Conference (ATC '18)*, pp. 789–794: https://www.usenix.org/system/files/conference/atc18/atc18-klimovic-serverless.pdf.

[6] A. Klimovic, H. Litz, and C. Kozyrakis, "Selecta: Heterogeneous Cloud Storage Configuration for Data Analytics," in *Proceedings of the USENIX Annual Technical Conference (ATC '18)*, pp. 759–773: https://www.usenix.org/system/files/conference/atc18/atc18-klimovic-selecta.pdf.

[7] P. Stuedi, A. Trivedi, J. Pfefferle, R. Stoica, B. Metzler, N. Ioannou, and I. Koltsidas, "Crail: A High-Performance I/O Architecture for Distributed Data Processing," *IEEE Data Engineering Bulletin* 40, pp. 38–49.

[8] Apache Crail (Incubating): https://crail.incubator.apache.org, 2018.

[9] A. Klimovic, H. Litz, and C. Kozyrakis, "ReFlex: Remote Flash ≈ Local Flash," in *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '17)*, pp. 345–359.

[10] A. Klimovic, Y. Wang, P. Stuedi, A. Trivedi, J. Pfefferle, C. Kozyrakis, "Pocket: Elastic Ephemeral Storage for Serverless Analytics," in *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI '18)*, pp. 427–444: https://www.usenix.org/system/files/osdi18-klimovic.pdf.

Noria
A New Take on Fast Web Application Backends

JON GJENGSET, MALTE SCHWARZKOPF, JONATHAN BEHRENS, LARA TIMBÓ ARAÚJO, MARTIN EK, EDDIE KOHLER, M. FRANS KAASHOEK, AND ROBERT MORRIS

Jon Gjengset is a Norwegian PhD student in the Parallel and Distributed Operating Systems group at MIT CSAIL. He received his bachelor's from Bond University, Australia, in 2011, and his master's from University College London in 2013. His primary research focus is on distributed data-flow systems, though he has also worked on computer security and wireless systems. Outside of academia, Jon develops teaching resources for the Rust programming language and is a frequent open-source contributor. jon@thesquareplanet.com

Malte Schwarzkopf is a postdoctoral associate in the PDOS (Parallel and Distributed Operating Systems) group at MIT CSAIL. His research focuses on distributed systems, with current and past work on data-flow systems, query compilers, cluster scheduling, datacenter networking, and parallel data processing. He received both his BA and PhD from the University of Cambridge, and his research has won best paper awards from EuroSys (2013) and NSDI (2015). malte@csail.mit.edu

Jonathan Behrens is a PhD student in the PDOS group at MIT. His research centers around operating systems and distributed systems, including Noria and work on OS abstractions to improve resource utilization. behrensj@mit.edu

Noria [2], first presented at OSDI '18, is a new web application backend that delivers the same fast reads as an in-memory cache in front of the database, but without the application having to manage the cache. Even better, Noria still accepts SQL queries and allows changes to the queries without extra effort, just like a database. Noria performs well: it serves up to 14M requests per second on a single server, and supports a 5x higher load than carefully hand-tuned queries issued to MySQL.

Writing web applications that tolerate high load is difficult. The reason is that the backend storage system that the application relies on—typically a relational database, like MySQL—can easily become a serious bottleneck with many clients. Each page view typically involves 10 or more database queries, which each take up CPU time on the database servers to evaluate. To avoid such slow database interactions and to reduce load on the database, applications often introduce caches (like memcached or Redis) that store already-computed query results for fast common case access. These caches, however, impose significant application complexity, because the application must query, invalidate, and maintain them [1]. Surely there has to be a better way.

Data-Flow for High Performance

At first glance, Noria seems similar to a database because it processes SQL queries. However, instead of evaluating queries on-the-fly as a traditional database would, the application registers long-term queries with Noria for repeated use. Queries contain free parameters that the application specifies when it actually executes its reads, similar to the interface provided by prepared SQL statements. From the pre-specified queries, Noria constructs a *data-flow graph* that *continuously* and *incrementally* evaluates the queries when the underlying data changes.

Data-flow processing was initially invented in the 1970s for circuit design but has recently been adopted for large-scale parallel data-processing in systems like Dryad [4], Naiad [5], and TensorFlow [6], for example. In data-flow, the system represents computations as a graph whose vertices are data-flow *operators* and whose edges carry *updates* between the operators. When an operator receives an update on an incoming edge, it processes the update (possibly consulting internal *state* that it keeps) and emits zero or more updates of its own on all its outgoing edges. This graph representation is appealing, as it makes the computation's dependencies explicit: update propagation across different edges and processing at different vertices can happen in parallel. Therefore, data-flow processing is well-suited to scaling across multiple CPU cores and servers.

In Noria, the data-flow graph connects classic database tables at its inputs to *materialized views* at its leaves. The intervening operators proactively execute the application's queries for each change to the tables. Noria generates the data-flow from SQL queries using a process similar to database query planning. Noria then serves all reads directly from the materialized views in memory, which makes reads as fast as reading from a cache. When the records in a

Lara Araújo is a Software Engineer at Airbnb working on large-scale distributed services and streaming data pipelines. Before joining Airbnb, Lara earned a bachelor's and a master's degree in EECS from MIT, focusing on developing secure datastores for high-performance applications. Her interests revolve around distributed systems and different kinds of storage systems. Lara was born and raised in Fortaleza and enjoys dancing, bouldering, and reading books by the ocean. lara.araujo@airbnb.com

Martin Ek studied computer science at MIT and the Norwegian University of Science and Technology. He's currently a Software Engineer at Stripe, where he helps build financial infrastructure for online businesses. mail@ekmartin.com

Eddie Kohler is a Professor of Computer Science at Harvard. He maintains (more or less) several widely used software packages, including the Click modular router and HotCRP, and he edited a musical score of John Cage's Indeterminacy. Twitter: @xexd, kohler@seas.harvard.edu

Frans Kaashoek is the Charles Piper Professor in MIT's EECS department and a member of CSAIL, where he co-leads the parallel and distributed operating systems group (https://pdos.csail.mit.edu/). Frans is a member of the National Academy of Engineering and the American Academy of Arts and Sciences, and is the recipient of the ACM SIGOPS Mark Weiser award and the 2010 ACM Prize in Computing. He was a co-founder of Sightpath, Inc. and Mazu Networks, Inc. His current research focuses on multicore operating systems and certification of system software. kaashoek@mit.edu

Robert Morris is a Professor of Computer Science at MIT. rtm@csail.mit.edu

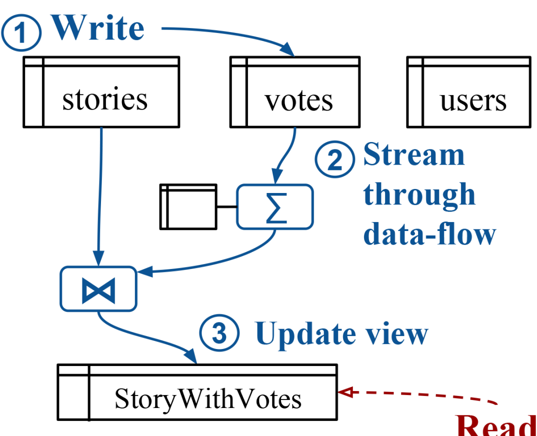


Figure 1a: Example Noria data-flow for a query that counts the votes for each story in a news aggregator and incrementally updates the count as new votes arrive (solid). Reads hit materialized view (dashed).

table change (e.g., in response to a client insert or update), Noria feeds updates through the data-flow to modify the materialized views as necessary.

The idea of materialized views has been around for decades, and some commercial and research databases support them. However, existing implementations lack the flexibility and performance that web applications require.

Noria's approach effectively flips the database query model on its head: instead of executing queries in response to reads, Noria executes them in response to *writes*. Reads are simple lookups into materialized state, which makes them (much) faster by moving work from reads to writes. Modern web applications are generally read-heavy, so this tradeoff makes sense for them. Furthermore, since Noria takes care of making reads fast even for complex SQL queries, the developer no longer needs to write error-prone, complex cache-maintenance code, or tune their queries for fast execution. They can simply issue the SQL queries they wish, inline aggregations and all, and Noria does the rest.

An Example: Votes for News Stories

Let's take a look at how Noria executes a particular SQL query. Figure 1a shows the data-flow that Noria constructs when given a query that counts the votes for each story in a news aggregator like Hacker News or Lobste.rs. The query joins with the `stories` table to retrieve the story's details (title, author, etc.). When a client inserts a new vote (let's say for the story with the identifier `A`), an update enters the data-flow at the vertex that corresponds to the `votes` table. From there, the data-flow propagates the update to the aggregation vertex below, which looks up the *current* vote count for the new vote's story in the internal state it maintains (say, 7). The count then updates the internal state to record that the vote count for that story is now 8 and emits an *update* to its children saying that the count for `A` is now 8, not 7. This update arrives at the join, which looks up `A`'s title in `stories` and produces a new update that says `A`, whose title is "Space elevator nearly completed," now has a vote count of 8, not 7. That update finds its way to the materialized view `StoryWithVotes`, which Noria updates appropriately so that any subsequent read from it sees `A`'s vote count as 8. Here, we say that `StoryWithVotes` is *keyed* by the story's identifier. In general, the key for a view is dictated by a set of free parameters in the corresponding SQL query issued by the application.

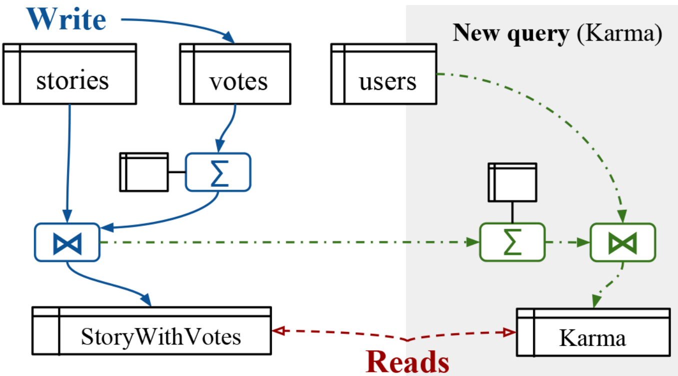


Figure 1b: If the application adds another query to compute the "Karma" score for each user (the total votes received for the user's stories), Noria dynamically adds to the running data-flow (dash-dot) the extra operators and materialized views needed.

Making Data-Flow Work for Web Applications

Naively adding new queries and initializing their data-flow state and materialized views may require Noria to compute a significant amount of state for the new query and induce downtime while it does so. More generally, if Noria always kept all state for all stateful internal data-flow operators and all its materialized views, its memory footprint would explode with many queries. Noria solves this problem by introducing *partially stateful* data-flow. This new model in turn enables Noria to support *dynamic* materialized views, where the set of queries changes over time without requiring a system restart.

Dynamic change. Figure 1b shows the data-flow from Figure 1a after the application adds a new `Karma` query (the shaded gray region). `Karma` computes the total votes for all stories posted by a given user. Notice that the data-flow path for `Karma` partially overlaps with that of `StoryWithVotes`. Noria realizes that it does not need to recount all the votes but can instead reuse the counts it already has. When the application first issues the `Karma` query, Noria extends the currently running data-flow to also include the extra data-flow operators needed for the new query and a new materialized view for `Karma`. It then initializes the state needed by stateful data-flow operators and the materialized view before making the latter available for application reads. Reads of old views are unaffected by changes to the data-flow, as are writes to unconnected parts of the data-flow. In combination with partial state, Noria makes the change instantaneous for writes as well.

Data-flow systems prior to Noria were designed for stream, graph, and parallel "big data" processing and cannot change the computation (i.e., queries) without restarting [6]. They must either keep all computed state in internal operator state and materialized views or apply *windowing* to reduce computed state by throwing away old records. For web applications, neither is acceptable: the backend cannot be down when queries change, and it must return complete results rather than ones based only on recent changes.

This brings us back to Noria's key idea: partially stateful data-flow. Noria's data-flow changes on-the-fly in response to query changes, and it keeps only a subset of state in memory, fetching missing data on-demand.

Partial state. Noria marks some keys in each data-flow state as *absent* and recomputes them only when needed. To support such recomputation—e.g., when a client reads an absent key from a materialized view—Noria relies on *upqueries* through the data-flow. Upqueries allow a vertex to ask its ancestors to recompute the absent state the vertex needs in order to serve an application read. The upstream ancestors respond to an upquery with the records in their state that match the absent key or keys specified by the upquery, and the results percolate back down through the data-flow. Since upqueries allow vertices to recover absent state, Noria is free to evict infrequently accessed state to save memory. More importantly, Noria also uses absent state to create new materialized views and operators with initially empty state, relying on upqueries to fill the state on demand. This allows Noria to adapt to most query changes entirely without downtime; all that is required is to bring up a set of empty data-flow operators. Absent state also speeds up regular processing, as updates for keys that are evicted, or that the application has never requested, can be discarded early.

Partial state and upqueries are conceptually simple, but making them always correct actually requires care. Intuitively, a partially stateful data-flow is only correct if it always—whether directly or via upqueries—produces the same result for a client read that a classic data-flow with full state would have returned. However, ensuring this in the face of concurrent processing in the data-flow, and with upqueries that can race with "normal" updates traveling downstream that themselves may be contained in the eventual upquery response, is difficult. Noria ensures this property using a new data-flow model and extra invariants. Some of the challenges are:

- How do data-flow operators handle updates that encounter absent state? Consider the earlier count: if its state for story `A` is absent, how can the count operator produce `(A, 8)` as the emitted update?
- How does parallel processing of complex data-flows that fork and join still ensure that upquery responses always contain all the updates processed at the queried operator exactly once?
- How do operators that change the key column handle upqueries? For example, the sum operator added in Figure 1 may upquery the join on its incoming edge for a particular user, but that join is keyed by the story identifier column.
- How do multi-ancestor operators handle upqueries if state for the upquery key is available in one ancestor but not in the other?

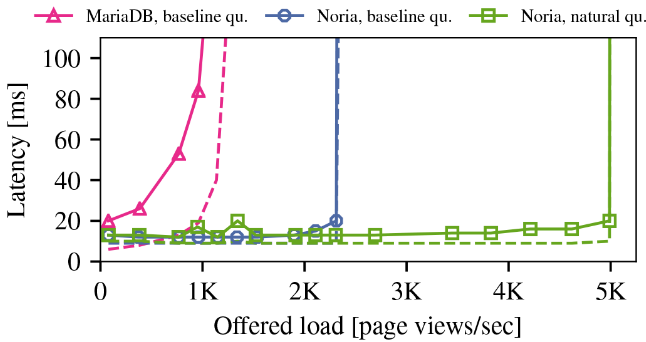


Figure 2: Noria scales to a 5x higher load than MySQL for the Lobste.rs website's workload while using queries free of hand-tuning (2.5x with the Lobste.rs's developers' original queries). Solid line shows median; dashed is the 95th percentile.

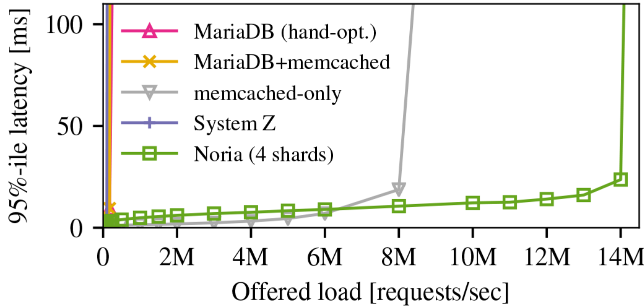


Figure 3: Noria supports 14 million requests/sec for a read-heavy (95% reads) workload, while other systems achieve only 200,000 requests/sec—with the exception of an unrealistic memcached-only setup that does strictly less work but still underperforms Noria.

Our paper [2] gives the invariants that Noria must maintain to guarantee correct execution and points out what goes wrong if these invariants are not properly maintained.

Evaluating the Noria Prototype

We implemented Noria in about 60,000 lines of Rust, along with a MySQL adapter that implements the MySQL binary protocol and makes Noria appear as a MySQL server to legacy applications. This way, Noria can support unmodified MySQL applications that use prepared statements (e.g., through PHP's PDO library). Noria supports sharding and partitioning the data-flow across cores and servers, and stores all base tables durably in RocksDB [7]. It handles failures in the distributed system by recreating those parts of the data-flow that a failure affects.

To evaluate Noria's performance and check that it *actually* makes web applications faster and reduces their complexity, we wrote a workload generator that emulates the real production workload seen by the news aggregator website Lobste.rs (https://lobste.rs). Lobste.rs is a Ruby-on-Rails application backed by a MySQL database, and the Lobste.rs developers carefully hand-optimized its queries for performance. Our benchmark issues the same SQL queries as the real Lobste.rs website, with the same frequency and popularity skew, using the MySQL binary protocol.

We then run that against both MySQL directly (we use MariaDB v10.1.34, a GPLv2 community fork of MySQL) and against Noria, on a 16-core Amazon EC2 VM. Figure 2 plots the offered load on the x-axis (in page views per second; each page issues around ten queries) and the achieved median and 95th percentile latency on the y-axis (so lower is better). At the point where each setup stops scaling—for example, because it saturates the server's CPU cores—the latency curve forms a "hockey stick" that shoots up when the system cannot keep up with the load anymore. The results indicate that Noria scales to a 2.5x–5x higher load than the MySQL baseline. For the initial result (blue line with circles, 2.5x improvement), we use the exact same queries as the Lobste.rs developers.

We then go a step further and remove all manual optimizations from the queries (line with squares). For example, the original application keeps `upvotes` and `downvotes` columns in the `stories` table and updates them on every vote, so that read query evaluation avoids doing a `COUNT` over `votes`. This is effectively a hand-rolled "materialized view" of the vote count, but it requires the developers to customize the application to update this column whenever the vote count changes. In Noria, such hand-tuning is unnecessary. Indeed, removing the hand-optimizations from the queries, we see a 5x speed-up over MySQL. The difference here comes from the fact that by not having to maintain these auxiliary values in the base tables (but instead having Noria maintain them in the data-flow), we avoid an extra `UPDATE` query and parallelize the update processing.

To quantify how much Noria improves performance over existing approaches, we choose a single, common query (the join of `stories` with vote counts) and issue that same query against a number of common web backend setups. Here, 95% of the requests are reads, and 5% are new votes, and we use a similar, skewed popularity distribution as the real Lobste.rs site observes. We benchmark MariaDB; System Z, a commercial database that supports materialized views; MariaDB with a memcached look-aside cache; "memcached-only," an unrealistic deployment where the application stores vote counts directly in memcached without any database interactions; and Noria with four-way sharding for parallel processing.

All systems run entirely in-memory to avoid measuring the I/O layer performance, and we set the databases to avoid transactions and use the lowest isolation level. Figure 3 again shows that Noria performs well: while the database-based systems do not scale beyond 200,000 requests/sec, Noria scales all the way to 14 million requests/sec. The unrealistic memcached-only deployment, for comparison, scales to 8 million requests/sec but then saturates the cores of the server.

Noria outperforms memcached because it uses a more efficient, lock-free data structure to serve reads, but this is not fundamental (memcached could use the same data structure). Noria's high performance comes because reads directly hit the materialized view, and because it processes writes efficiently through the sharded, partially stateful, incremental data-flow.

When to Use Noria

Noria is designed for web applications that are read-heavy and that can tolerate eventual consistency. The ubiquity of caches in modern web application stacks suggest that eventual consistency is often sufficient, although we are also working on ideas for high-performance transactions on Noria. Noria also obviates the need for transactions in some cases. The Lobste.rs developers, for example, only use transactions to ensure that a story's vote count is incremented atomically with the vote being stored. Noria maintains the vote count internally in the data-flow, so this transaction is no longer necessary.

Noria primarily targets applications whose working set fits in memory when sharded and partitioned across many servers. Old records in base tables are only on disk, but applications that regularly need to access the full data set (e.g., full-text search) would need additional support to work well in Noria.

How to Use Noria

Noria is open-source and available at https://pdos.csail.mit.edu/noria. In many cases, you should only need to start up the Noria MySQL adapter, point your application at it instead of MySQL, and turn off all your caches. Noria will take care of the rest. The Noria prototype is research code and still in development, but we would like to hear how it works for other people!

References

[1]: J. Mertz and I. Nunes, "Understanding Application-Level Caching in Web Applications: A Comprehensive Introduction and Survey of State-of-the-Art Approaches," in *ACM Computing Surveys*, vol. 50, no. 6 (November 2017), pp. 98:1–98:34.

[2]: J. Gjengset, M. Schwarzkopf, J. Behrens, L. T. Araújo, M. Ek, E. Kohler, M. F. Kaashoek, and R. Morris, "Noria: Dynamic, Partially-Stateful Data-Flow for High-Performance Web Applications," in *Proceedings of 13th USENIX Conference on Operating Systems Design and Implementation (OSDI '18)*, pp. 213–231: https://www.usenix.org/conference/osdi18/presentation/gjengset.

[3] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani, "Scaling Memcache at Facebook," in *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation (NSDI '13)*, pp. 385–398: https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/nishtala.

[4] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks," *SIGOPS Operating Systems Review*, vol. 41, no. 3 (March 2007), pp. 59–72.

[5] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi, "Naiad: A Timely Dataflow System," in *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP '13)*, pp. 439–455.

[6] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: A System for Large-Scale Machine Learning," in *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI '16)*. pp. 265–283: https://www.usenix.org/system/files/conference/osdi16/osdi16-abadi.pdf.

[7] RocksDB: https://rocksdb.org/.

SRE

Achieving Reliability with Boring Technology

DAVE MANGOT

Dave Mangot is the author of *Mastering DevOps* from Packt Publishing. Previously, he led site reliability engineering (SRE) for the SolarWinds cloud companies. An accomplished Systems Engineer with over 20 years' experience, he has held positions in various organizations, from small startups to multinational corporations such as Cable & Wireless and Salesforce, from Systems Administrator to Architect. He has led transformations at multiple companies in operational maturity and in a deeper adherence to DevOps thinking. He enjoys time spent as a mentor, speaker, and student to so many talented members of the community. usenix@mangot.com

Everything should be made as simple as possible, but no simpler.—*Albert Einstein*

Distributed systems. Complex systems. Enterprise systems. No matter how we're involved in computing these days, it's likely we're working on complex or complicated (in the Cynefin sense) problems. In fact, even systems that start out simple ultimately become complex through the continuing evolution of those systems through architecture changes, code deploys, or simply the passage of time (do you remember why you made that choice three years prior?). Because this complexity is a naturally occurring property of these systems, I choose to use boring technology.

When I say "boring technology," we should give credit to one of its biggest proponents, Dan McKinley (@mcfunley) who wrote: "We should generally pick the smallest set of tech that covers our problem domain, and lets us get the job done" [1].

Why do I feel this way? I've been doing Operations work for more than 20 years. I've worked in small startups and big multinationals. I've worked on huge monoliths and systems that had an undying allegiance to services. Through it all, I've encountered complexity. When it's 3 a.m. and the pager is blowing up, complexity is not my friend (or yours). Over the years, I've always tried to advocate for the "smallest set of tech that covers our problem domain." When you're firefighting and you're trying to reason about what is wrong, why the Java process keeps OOM'ing or why the database connection pool is being exhausted, the last thing you want is fancy, magical, technology.

MTTR > MTBF

You may be able to tell, I have a specific bias to the Operations perspective. As site reliability engineering (SRE) has become more prevalent, we can see an emphasis on reliability and recovery from failure. In an ideal world, our recovery from failure is instantaneous; the customer has no idea there was a failure. Unfortunately, we don't live in an ideal world, so the best we can do is to try to minimize downtime by maximizing our ability to recover from failure. Choosing boring technology is a proven technique for making this a reality.

Does your Operations (DevOps, SRE, etc.) really need to deploy multiple Kubernetes clusters in order to deploy a single Ruby script? Should we try to write our next service in Erlang because we heard it's "cool," even though our staff mostly consists of PHP programmers? Boring technology works well for us because we have more ability to reason about it. If my ability to form a mental model of the system I'm working on is hampered by my inability to understand the technology, either because of complexity or obscurity, I'm going to have a bad time.

Often the main problems with fancy technology is that it is optimized to try to prevent failures, not recover from failures. Many fancy "enterprise" technologies are created in this way. One way to think about this is in terms of horizontal vs. vertical scaling. You are probably in good shape if your solution is designed to scale horizontally, where the loss of any single component is easily handled by other easily replaceable components with no noticeable effect to the customer. If your solution is designed with multiple somethings (power supplies, network cards, etc.) within a single component, you may be relying on fancy technology. If you lose

one of those systems, where does that leave you? Systems that optimize for mean time between failures (MTBF) instead of mean time to recovery (MTTR) are prone to what author Nasim Taleb calls "black swan events":

> [T]he problem with artificially suppressed volatility is not just that the system tends to become extremely fragile; it is that, at the same time, it exhibits no *visible* risks…These artificially constrained systems become prone to Black Swans. Such environments eventually experience massive blowups…catching everyone off guard and undoing years of stability or, in almost all cases, ending up far worse than they were in their initial volatile state. [2]

MTBF-Optimized Infrastructure

What are some examples of complexity evident in MTBF-optimized infrastructure? Have you ever configured network bonding on a Linux host? How many different modes are there for bonding? Six. That means that there are six different ways that you could possibly expect that your systems will behave in the event a network interface is lost. To what end? Well, to protect us from the case where a system could potentially disappear off the network. But is a NIC failure really the only way a system could disappear off the network? What about power supply failures? What about running out of memory or CPU? What about file system corruption? How many different components do we want to make redundant in order to guard against a system disappearing off the network? How much do we want to pay for those systems? Can we really foresee all possible failure scenarios?

What if we were to think about it a different way? What if we *expected* that systems would disappear off the network? If we design our systems in this way, we're *protected* from systems disappearing no matter *what* the reason! Additionally, because I'm spending less money per system, I can usually have more of them for the same cost. This increases my ability to tolerate failure, even *multiple* failures. This is another problem we often see when we try to choose fancy enterprise systems with multiple layers of complex protection within a single system. We can't afford many of the components, and thus we are often left with only two of something, a primary and a backup. Not only is this very inefficient (we've paid a lot of money for a system that most of the time does absolutely nothing), but in the event of a failure, we're now one failure away from catastrophic failure. Additionally, we're subject to relying on all that other money we spent on our enterprise support contract to deliver the necessary part on the 12x5 or 24x7 guaranteed response times as offered by the vendor. If the vendor doesn't have the part, or the power spike that blew out the first system comes back, we could be in a very bad situation.

Cattle vs. Pets

Instead, we should choose boring technology. If a system goes down, the load balancer stops sending it traffic because it's failed its health check, and we replace it with an exact replica. We don't care about an individual system, we care about the overall system. Many of you have probably heard of this as cattle vs. pets [3].

If a pet gets sick, we do what we can to make it better (like calling in enterprise support). If a head of cattle gets sick, we worry about the overall health of the herd. While we can't as readily replace one head of cattle, we can readily replace a server, especially in cloud or cloud-like environments.

As our systems mature and grow, we often see the wisdom of being able to control and reason about them in simple ways. This use of boring technology doesn't just have to apply to application servers, it can apply to networking or storage as well. Let's look at some examples.

Networking

If we were to look up the DNS information for www.atlassian.com (this is just one example), we would notice something interesting.

```
$ host www.atlassian.com
www.atlassian.com is an alias for pledge-vtm-ash2-prod
-public-01.atlassian.com.
pledge-vtm-ash2-prod-public-01.atlassian.com is an alias for
pe-vt-vtmnl-1h5icdrzt7xcp-d84e3144685e1b8d.elb.us-east-1
.amazonaws.com.
pe-vt-vtmnl-1h5icdrzt7xcp-d84e3144685e1b8d.elb.us-east-1
.amazonaws.com has address 18.234.32.152
pe-vt-vtmnl-1h5icdrzt7xcp-d84e3144685e1b8d.elb.us-east-1
.amazonaws.com has address 18.234.32.153
pe-vt-vtmnl-1h5icdrzt7xcp-d84e3144685e1b8d.elb.us-east-1
.amazonaws.com has address 18.234.32.154
```

Three IP addresses! That's strange! If you've ever spent any time with enterprise-grade networking gear, there is often a "floating IP" that can bounce back and forth between two pieces of equipment depending on which is currently responsible for handling the traffic (and the other sits idle, despite the fact that we've paid for it, just in case). That IP address would be presented to the world as a single IP. But in this case, we have three. Why? Because Amazon has the ability to replace components of its load balancers and actually does this with a fair amount of regularity. When they need to upgrade or replace a piece of hardware or software, they don't exercise the HSRP or VRRP sequence for shifting traffic to the "other" host. They replace the component itself, like cattle.

Storage

Solving a problem like storage at the level of Facebook could be a daunting challenge. If you needed to store all those baby pictures, profile pictures, wedding pictures, etc., that could be a tough problem. If you were Facebook, you may have started out using a number of enterprise class (or Pet) solutions. As a matter of fact, this was actually the case, until Haystack [4]. You can read the paper yourself, but this is from the conclusion: "Haystack provides a fault-tolerant and simple solution to photo storage at dramatically less cost and higher throughput than a traditional approach using NAS appliances. Furthermore, Haystack is incrementally scalable, a necessary quality as our users upload hundreds of millions of photos each week." Moving to a simple solution for the win.

Making Change

This idea of choosing simple (boring) solutions that we can reason about more easily may sound appealing at this point. But how do we make these changes in our existing organizations? How do we get to a point where we have simple recovery that we know both works and is well tested and practiced? As Gene Kim says of DevOps in "The Three Ways" [5], "repetition and practice is the prerequisite to mastery."

Just as Facebook was happy that their solution was incrementally scalable, the happiest path to making these kinds of changes is incremental as well. While we'd all love to have Netflix's Chaos Monkey running in our infrastructure tomorrow, proving all is well, that's as unrealistic as standing up a shiny new Kubernetes cluster tomorrow and understanding how to deploy and operate it. My favorite method for making change is what we often call *Crawl-Walk-Run*.

Crawl-Walk-Run

We are not born with the ability to run. There is a progression we must go through in order to reach that level of mastery (which takes repetition and practice!). So it is with maturation of processes or architecture when we are adopting boring technology.

Crawl

So how do we get started? How can we "crawl" when moving from our fancy enterprise technologies to something simpler? The first step is to configure just about everything with code. When we say everything, we mean Docker containers, servers, network gear, RAID cards, etc. We are trying to configure everything this way. This gives us a number of advantages:

- If we're doing infrastructure as code, we can version things, because they are in revision control. That means if I ever want to know how something was configured on March 22nd, I can look that up.
- That ability also gives me the ability to create representative test environments and have confidence that those environments are configured in the same way as production. If my test fails in a representative test environment, I have high confidence it would have failed in production.
- I also have confidence that any time I have a component of type X, it will be configured identically to every other component of type X with the "push of a button." One need look no further than the Knight Capital failure [6] to recognize the dangers of having differently configured systems that are supposed to be identical. Reasoning about multiple possible configurations of the same component interacting with each other is extremely difficult! Remember our Amazon load balancer example? Every time a load balancer component is swapped out, Amazon knows exactly how the new component will be configured. Every time a new Haystack node is deployed at Facebook, they know exactly how it will be configured.

There are many ways to configure things as code. We have configuration management tools, and we have config files or settings that can be checked into repositories. We can even use things like Puppet types and providers to interact with our RAID cards or out-of-band management cards to make sure they are configured perfectly every time. Many network vendors are now offering APIs we can interact with for our network gear to make sure they are configured properly.

If your fancy piece of tech does not offer a programmatic way of configuration, you are probably not using boring technology and have something designed to be manipulated by the messy bags of mostly water we call humans. Eliminate those from your infrastructure—the component, not the humans!

Walk

Now that we have confidence that our infrastructure will be configured properly each and every time (how quickly could you rebuild a server that was removed with an exact replica?), we are ready to experiment with failure. One relatively easy way to accomplish this is with production readiness game days.

In this scenario, before we allow a new service or major infrastructure component to be deployed to production, we test it to learn about failure. How does it fail? What is impacted? How do we even know it's failed? How do we recover?

If repetition and practice are the prerequisite to mastery, then we need to have an opportunity for repetition and practice. We do this by making a test plan of exactly what we will fail (in our representative test environment) and what the expected behavior will be. Maybe we will block the DNS servers. Maybe we will pull a disk. Maybe we will terminate an instance. There are many options. We also need to determine where the test data

will come from if required. Copying production data can have security implications. Can we use synthetic data? This plan should be agreed upon by all the parties involved (Dev, Ops, DBA, etc.). Then the plan should be executed. This has a number of advantages:

- No complex systems can ever be "thrown over the wall" to Operations for deployment. If there is an unexpected behavior during the failure scenarios, the party responsible for fixing that behavior will be given as many opportunities as necessary to fix the offending behavior until the game day is declared successful.
- The folks responsible for remediating failure will have the opportunity to practice those remediations! No one wants the first time they attempt to recover a failed system to also be the first time anyone has ever attempted to recover said system. By practicing before production, you have the opportunity to not only learn how to do it, but to also ask for clarification, make suggestions, improve documentation, etc.
- We can often discover unintended consequences of the deployment of the new system. This is why representative test environments are so important. We don't want to discover that our database would run out of connections the first time the system is activated in production.
- It reinforces the idea that the availability of our production systems is everybody's responsibility. Not just the people who will be woken up in the middle of the night, but the entire team responsible for delivery of that component of the infrastructure.
- It gives us an opportunity to find out where our technology is not boring. If, during the game days, we repeatedly have problems restoring our systems to the proper state, or understanding the failure scenarios, maybe our system is not quite as boring as we thought. That is an opportunity to revisit the design, and the choices made, and make the necessary adjustments so that we can eliminate single points of failure, fancy vendor solutions that never quite live up to their promise, or that configuration that everyone could have sworn was in revision control but in fact was only placed as an unintended side effect of some other process.

Run

Once we've settled on our boring technology, and have confidence in our infrastructure and ability to detect and remediate failures, it's time to make that a regular part of how we operate. Both in participating more regularly in the design phase as well as after the system is deployed. This is a great time to get started with chaos engineering, a natural progression from the use of boring technology.

As Nora Jones said at ReDeploy 2018, "Chaos Engineering isn't done to cause problems; it is done to reveal them" [7]. We already know that our systems become more complex over time and that the system that we deployed two years ago has changed or morphed over time into something that can have many different properties than it did when first deployed. How do we ensure we can still recover from failures? By continually testing the infrastructure to make sure that the result of failures continues to be as we expect.

The problems that we will experience in production will become problems because complexity is an emergent property of these systems. If we expose those problems under controlled circumstances (people in the office at their desks, only one variable changed at a time, etc.), we will have a much higher likelihood of being able to detect and recover quickly, and then work to prevent those problems in the future. If we have these problems but don't reveal them, then we are setting ourselves up for Taleb's black swan events that can "catch everyone off guard" and "undo years of stability." That doesn't sound very boring to me!

Conclusion

When working in our professional roles as SREs, or storage administrators, or network engineers, etc., we are often heavily invested in the technology choices we make. Sometimes we may want to use some new technology because it's got a great reputation or because a lot of other people are using it. If it is not a technology that we understand well, or have the ability to understand well, we can often make choices that will cause us more problems down the road.

For that reason, when facing these choices, it is good to remember to choose boring technology. The complexity will be there, there is no running away from that. The systems will grow more and more complicated until it's time for that big refactor, which is a recognition that our systems are no longer boring but, rather, are collapsing under their own weight of complexity.

But there are ways to minimize those conditions and for us to mature our way out of bad situations when we find that we are in one. Choose boring technology.

References

[1] http://boringtechnology.club/.

[2] N. Taleb, *Antifragile: Things That Gain from Disorder* (Random House, 2012), p. 105.

[3] R. Bias, "The History of Pets vs Cattle and How to Use the Analogy Properly," September 29, 2016: http://cloudscaling.com/blog/cloud-computing/the-history-of-pets-vs-cattle/.

[4] D. Beaver, S. Kumar, H. C. Li, J. Sobel, P. Vajgel, "Finding a Needle in Haystack: Facebook's Photo Storage," 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI '10): https://www.usenix.org/legacy/event/osdi10/tech/full_papers/Beaver.pdf.

[5] G. Kim, "The Three Ways: The Principles Underpinning DevOps," 2012: https://itrevolution.com/the-three-ways-principles-underpinning-devops/.

[6] D. Seven, "Knightmare: A DevOps Cautionary Tale": https://dougseven.com/2014/04/17/knightmare-a-devops-cautionary-tale/.

[7] N. Jones, "Chaos Engineering: A Step Towards Resilience": https://youtu.be/qyzymLlj9ag?t=399.

Save the Date!

OpML '19 **2019 USENIX Conference on Operational Machine Learning**

May 20, 2019 • Santa Clara, CA, USA

The 2019 USENIX Conference on Operational Machine Learning (OpML '19) provides a forum for both researchers and industry practitioners to develop and bring impactful research advances and cutting edge solutions to the pervasive challenges of ML production lifecycle management. ML production lifecycle is a necessity for wide-scale adoption and deployment of machine learning and deep learning across industries and for businesses to benefit from the core ML algorithms and research advances.

Program Co-Chairs:
Bharath Ramsundar, *Computable*
Nisha Talagala, *ParallelM*

www.usenix.org/opml19

Anticipating and Dealing with Operational Debt

LAURA NOLAN

Laura Nolan's background is in site reliability engineering, software engineering, distributed systems, and computer science. She wrote the "Managing Critical State" chapter in the O'Reilly *Site Reliability Engineering* book and was co-chair of SREcon18 Europe/Middle East/Africa. Laura is currently enjoying a well-earned sabbatical (and tinkering with some of her own projects) after 15 years in industry, most recently at Google.
laura.nolan@gmail.com

We are all familiar with the concept of technical debt, the idea that over time, software systems become harder to change and maintain because of shortcuts taken earlier. An example of technical debt is the lack of a comprehensive suite of unit tests (or a flaky test suite). Old, unused code that hasn't been removed is another example. Technical debt can occur early in a system's lifespan as shortcuts are taken in order to launch, but most of the time the problem of technical debt gets worse as a system ages.

Operational debt is different. It happens when a system is launched, or experiences rapid growth in usage, before operational tasks are automated, leaving them to the system's human operators. In organizations with a focus on automating routine operational tasks, much of this is solved over time, leading to less operational debt as a system ages.

Technical debt is like credit card debt—acquired piecemeal over time. Operational debt is more like a mortgage: it can be paid down over time leading to ownership of a stable, well-automated system. However, sometimes people do have problems paying off their mortgages. The worst case scenario is a team that ends up with so much operational debt that they don't have cycles to work on fixing it, instead spending most of their time on *toil* [1]—tactical work that doesn't improve their systems in the long term.

This environment isn't good for engineers, and teams in this situation will struggle to retain staff.

Types of Operational Debt

There are five main categories of work that, if not automated, lead to operational debt.

One is routine housekeeping that happens on a schedule. This might include taking and validating backups, updating certificates, and making sure personally identifiable information is deleted after a certain time period.

The second is recovery from routine failures like loss of a hard drive, transient network problems, or a machine restarting.

Another category involves managing change over time. This includes things like performing migrations, doing capacity planning, and monitoring for performance regressions.

Many systems involve some routine per-customer work like setting up permissions, quotas, or other resources.

Finally, there is non-routine work that scales with your system's growth. This includes turning up new instances of your systems, dealing with abusive users, resharding datastores to deal with growth, and investigating performance issues on behalf of customers.

Operational Debt after Launch

Some amount of operational debt in a newly launched system is inevitable. This is for two major reasons.

The first is unknown unknowns—issues will crop up in production that weren't anticipated, and some of these will need automation to handle them. For example, take a system where the underlying datastore occasionally has replication issues. Sometimes a customer's changes don't get reflected everywhere, they complain, and someone has to go and unwedge it by hand. There are several potential approaches to automating this problem away, ranging from fixing the underlying replication issues to various bolt-on approaches, but either way, noticing the pattern and automating away this sort of problem takes time.

The second reason is that even for routine and anticipated operational tasks, the development of the core system itself usually has to precede development of complex automation. It's hard to automate a process for a system that doesn't exist yet.

Managing and Planning for Operational Debt

Operational debt is not inherently bad, but too much of it certainly is—again, like mortgage debt. It needs to be planned for and managed, particularly when launching a new service, instituting a major change to an existing service, or in times of fast growth.

First, track what your team is spending its time on now. If your team already has a lot of operational work, it may need to be reduced before you can afford to launch something new. At Google, SRE teams aim to spend under 50% of their time on operational work.

Next, estimate what you're getting yourself into. For your planned system or feature:

- What are the periodic "housekeeping" tasks?
- What failures or problems will the system likely encounter regularly, and how much work will it be to recover from them?
- What are the change management tasks?
- What are the routine per-customer tasks and the likely non-routine ones?
- How is the user base likely to grow over time?
- What is automated already, and how much effort is likely to be required to automate the rest?

You should also budget for some unknown unknowns. This is technology, after all.

After this exercise, you should have a better idea whether or not your team will be able to afford the launch and what needs to be automated first so that your team can remain productive.

Zero operational work shouldn't be your goal. Some tasks aren't worth automating because they're done infrequently and there won't be a positive return on the investment of time. Some operational work is novel, like debugging new problems and dealing with outages, and does require human skills. But excessive operational debt is dangerous when it soaks up so many of a team's cycles that they can't do engineering work.

Anticipate operational debt, budget for it, and keep your team out of operational overload [2].

References

[1] V. Rau, "Eliminating Toil," in *Site Reliability Engineering (O'Reilly, 2016)*: https://landing.google.com/sre/sre-book/chapters/eliminating-toil/.

[2] R. Bosetti, "Embedding an SRE to Recover from Operational Overload," in *Site Reliability Engineering (O'Reilly, 2016)*: https://landing.google.com/sre/sre-book/chapters/operational-overload/.

PROGRAMMING

How to Reinvent the Bicycle

SERGEY BABKIN

Sergey Babkin has been employed as a Software Engineer for well over 20 years. His work experience includes SCO, Sybase, Microsoft, and, currently, Google. He likes to analyze and improve things. sab123@hotmail.com

Using programming puzzles as part of job applicant interviews has become common practice. While interviewing applicants, I've noticed two patterns in how they go about solving these puzzles. In this article, I examine these patterns and detail how programmers in general need to problem solve using the best of both patterns.

The Intuition and the System

In conducting recent job interviews, I've met a spate of junior engineer candidates with a similar issue: they quickly come up with a pretty good overall idea of the solution to a problem, and they can write code, but they fail to translate their solution into the code. They couldn't seem to organize the overall idea into components and then, step by step, work through the details and interdependencies of those components and sub-components, even with intense hinting from my side.

A bigger problem can always be seen as being composed of smaller, easier problems. The easier problems aren't necessarily easy, but two methods in dealing with them can be helpful: First, you can subdivide them further into even simpler problems. Second, as you try to solve a problem, you can gain an understanding of why it's difficult, and this often provides insight into solving the problem by avoiding it rather than overcoming it, by subdividing its parent problem differently. Not that all problems can be avoided: some things have to be overcome. The job applicants could come up with good ideas that solved difficult things that needed to be overcome, but they couldn't build a structure for the whole solution, where they could put these good ideas to good use.

To illustrate through an analogy, some time ago I read about an artist who would ask people to draw a bicycle from memory and then produce, as an art object, a bicycle based on the drawing. The results were art objects because they were completely non-functional. If I were to draw a bicycle without thinking, I would also produce something like that.

By spending some thought, any engineer should be able to reproduce a proper bicycle from the general logic: the function of the main components (wheels, steering, seat, pedals, chain) and the general considerations of the strength of the frame that shape it. The same logic can be used to check that none of the main components were forgotten: for example, if you forget about the chain, the pedals would be left disconnected from the rear wheel, so you'd have to remember it. Each component might be very non-trivial (the said chain took a long time to invent), but once you know the components, it should be impossible to put them in the wrong place.

This is something that should be done almost mechanically, with little mental effort. And yet these programming candidates could not do it. They tried to do it by intuition, but their intuition was not strong enough to handle a complex problem in one gulp, and they didn't know how to use the systematic approach either. The hints didn't help much; they didn't cause the right systematic associations.

Two Skills

There are really two orthogonal skills involved in solving these problems: to imagine the whole solution using highly developed intuition; to subdivide the problem and work through it iteratively, backtracking as necessary. Both are required to be a good engineer. A simple problem can be solved by using either of these skills alone. But even a moderately complex problem requires both skills; it's too big for intuition to figure out every detail, and too non-obvious for the systematic approach to find a good result in any reasonable time.

The problem I ask is actually quite difficult, too difficult for a perfectly adequate junior-to-mid-level engineer, and I'm not sure if I myself would have solved it well some 20 years ago. I know that I can solve it now, as it came from my real-life experience where I had to solve it really quickly from scratch. So I don't expect a good solution from this category of candidates; for them, a so-so solution is plenty good enough. Some of them actually do very well, producing a fully completed optimal solution.

There is a marked difference in how people with the one-sided development solve it, depending on which skill is their strong one. People with poor intuition and strong systematics produce a complete solution that is not very good. People with strong intuition and poor systematics get the right overall idea, figuring out the conceptual parts that I consider difficult and important (that the systematic group never figures out), only to fail miserably to work out all the details necessary to write the code. Not that the code doesn't get produced at all (though sometimes it doesn't), but what gets produced is closer to being an art object than working code.

Intuition, the Harder Skill

And that feels like a shame, because intuition is usually considered the harder skill to develop, requiring more time for development and being more rooted in natural ability. So there are people who could be good engineers if only they learned how to work systematically.

The trouble, I think, is that people are not really taught to do this kind of thinking in programming. Books and college courses describe the syntax of programming languages and the general picture but leave a void between these layers. People may learn this on their own from examples and practice. But the examples and practice tend to train the intuition, and people are left to figure out the systematic approach on their own, and they either figure it out or they don't. It looks like quite a few of the generally smart people either don't or take a long time to develop it. Yes, there are descriptions of how a problem has to be divided into the smaller parts, but they tend to miss the backtracking and the iterative redesign, making it look like intuition produces the right subdivision in one go. Not to say that there is anything wrong with intuition, it's my favorite thing, but the systematic approach allows you to stretch a good deal beyond the immediate reach of intuition, and to strengthen future intuition.

I've recently seen a question on Quora—"As you gain more experience, do you still write code that works but you don't know why?"—and this I think is exactly the difference between the intuitive and systematic solutions. Intuition might give you some code that works, or that possibly doesn't. The systematic approach lets you verify that what the intuition provided actually does what it's supposed to do and provides the stepping stones for the intuition to go further, both to fix what is going wrong and to produce more complex multi-leap designs.

Programming is not the only area with this kind of teaching problem. I think math has the same issue. The way proofs of various theorems are taught is usually not how the authors originally discovered them. These proofs get edited and adjusted a lot to make them look easier to understand. But then the teaching aspect of how to create new proofs through systematic trial and error gets lost.

Teaching the Two Skills

So how would you teach it? The bicycle example suggests that there is probably a general transferable skill too, and this skill can be trained by puzzle games like the classic "The Incredible Machine," where the goal is to build a Rube Goldberg contraption to accomplish the particular goal from a set of components. As in real life, the tasks there might include the extra components that look useful but don't really work out, or provide multiple ways to reach the goal. This of course requires that you achieve only one exact goal, while in programming you have to solve a whole class of related goals that include the corner cases. But this still might be a good place to start.

Perhaps the way to do it for programming is by walking through the solutions of complex problems, showing step by step how you can try the different approaches, follow through their elements, try to resolve the observed issues, and use this newly gained experience to find easier approaches. There are books built around somewhat different but closely related ideas: *Programming Pearls* and *More Programming Pearls* by Jon Bentley come to mind. *The Practice of Programming* by Brian Kernighan and Rob Pike, and, dare I say, my own *The Practice of Parallel Programming* are other examples.

A Systematic Puzzle

To give an example of what I think needs to be taught, I've decided to create a programming puzzle based on another, simpler interview problem that I used to use. The required insights in that problem are much smaller; it's much more about the systematic approach.

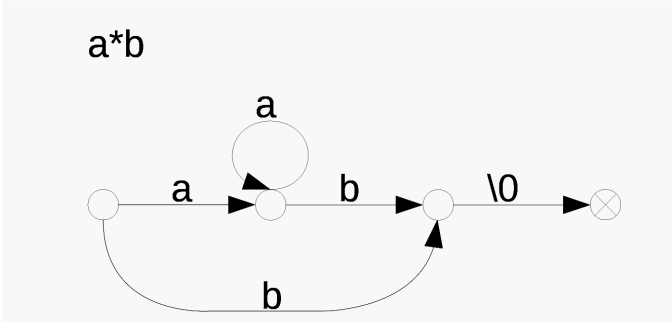


Figure 1: Finite state machine (FSM) for matching “a*b”

Since blindly remembering the solution to the problem is of no use to anyone, I want instead to show how better solutions can be born out of bad solutions. And it’s not just brute force versus some ingenious algorithm. All the solutions to this problem are essentially brute force, but some of them are better and simpler than the others.

I’m going to start with the worst solution I can think of and then gradually show the better solutions. The puzzle for you, the reader, is to use the difficulties in these solutions as hints towards better solutions that would take you as far ahead as possible.

I wrote those solutions as I would do at an interview, without actually compiling and running the code on a computer, so they might contain bugs, but hopefully not many bad ones.

The problem is to write a matcher for the very simple regular expressions, that include only the operators “.” (any character) and “*” (zero or more repetitions of the previous character). The “*” is greedy, consuming as many matching characters as possible. There is no escape character like backslash. The string must match completely, as if the regexp implicitly had anchors like “^” at the front and “$” at the end. And let’s say that the string is in plain ASCII, so we don’t need to bother with the wide characters.

The function declaration in plain C will be:

```
int match(const char *pattern, const char *text);
```

It will return 1 if the string matched the pattern and 0 if it didn’t.

Let’s start with the analysis. The first thing to notice about this problem is that some patterns in it are impossible to match. The “a*a” will never match anything because the greedy “a*” will consume all the “a”s, and the second “a” will never encounter a match. The same goes for “.*” followed by anything, because “.*” will consume everything to the end of the string.

The first solution proceeds in the most complicated way I can think of. You might have attended a college course on parsing that talked about the finite machine matcher for regular expressions. The most unsophisticated approach is to push this way blindly.

Figure 2: FSM for matching “.*b”

Before doing a finite machine, you’d really need to think of the state machine graphs you would be building for various regular expressions. I really could not get this code right until I had drawn the graphs.

Here are some examples: “a*b” is shown in Figure 1.

“.*b” (with “any” meaning “everything but \0”) is shown in Figure 2. This graph would never match anything, because it would never get into the final state (X). The FSM for “a*b*c” is shown in Figure 3, and “a*.*” in Figure 4.

Each state node of the finite machine graph would be represented by a dynamically allocated structure that has a plain array of the possible exits from that node, one per each character, and a flag showing that this node is final.

```
struct Node {
   Node *exits[256];
   int final;
};
```

The \0 could be handled as one of the normal exits, pointing to the final node. But there really isn’t much point in having a separate node just to carry the final flag. It’s easier to just set the final flag directly on a node that accepts an \0.

The graphs then become simpler, the graph in Figure 4 becoming as shown in Figure 5.

Since we’re dynamically allocating the nodes, we need to take care of freeing them too. And that means taking care of keeping track of them while we use them. The inter-node links are no good for this purpose, since they branch multiple ways, and some graphs might even have some disconnected parts. But we can notice that there would always be as many nodes as elements (plain letters or starred letters) in the pattern, plus one. So we can just allocate the nodes as a single array and then free them as a single array.

This is a good time to stop and think about the question, is there really any point in bothering with the nodes? They will be strung generally sequentially, just like the original pattern. So why not just use the pattern directly? Indeed, this is a simpler approach. Time to change gears.

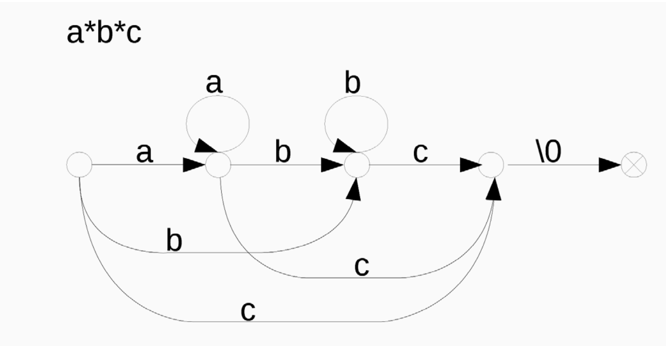


Figure 3: FSM for matching "a*b*c"

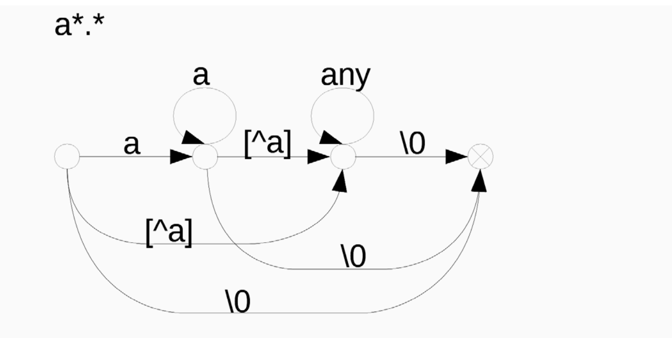


Figure 4: FSM for matching "a*.*"

Matching directly by pattern also has harder and easier versions. Again, let's start with the harder version.

The loop will be working in very much the same way as the matching loop in the parsed-pattern version (as some textbooks would teach) but will read the pattern directly from the string as it goes along.

Before writing the code, let's talk through the logic: as we read the next character of the text, we have a pointer to the next pattern element to parse. We parse the pattern element and match the text character to it. If the element is \0, we accept \0 and stop. If the element is starred and the character matches, we return the pattern back to the original position. If the element is starred and the character doesn't match, we try the next element from the pattern. If the element is '.', we accept everything but \0. If the element is another character, we accept it literally.

```
bool match(const char *pattern, const char *text) {
   char last_pattern = '\0';
   const char *p = pattern;
   for (const char *t = text; ; t++) {
      while (true) { // for starred sequences in pattern
         char element = *p++;
         if (element == '\0') {
            return *t == '\0';
         }

         if (*p == '*') {
            if (element == '.' && *t != '\0'
            || *t == element) { // matched
               --p; // return to the start of current element
               break;
            }

            // consume the star before reading the next element
            p++;
            continue;
         }
         if (element == '.' && *t == '\0'
         || *t != element) { // didn't match
            return false;
         }
         break;
      }
   }

   return false; // never reached
}
```

The inner loop is necessary to handle the sequences of multiple starred characters, such as "a*b*c" matching the "c". If we don't do the loop, "c" would get compared to "a", and the match will be considered failed.

The outer "for" loop here is interesting, without an exit condition. This is because the '\0' is matched inside the inner loop mostly in the same way as the normal characters: `(*t != element)` handles the unexpected '\0' in the same way as any other unexpected character. It's easy to start writing the loop with:

```
for (const char *t = text; *t != '\0'; t++)  {
   ...
}
return element == '\0';
```

But that would miss the situation where the pattern ends with a sequence of starred characters. This is something that is easy to miss, but it would be detected by a careful code analysis, a good unit test, or by a helpful interviewer. Then the code would need to be fixed by either bringing the handling of '\0' entirely into the inner loop as I have done here (there is no reason to be afraid of the loops that look nonstandard, they can be quite useful) or by moving the inner loop into a function and calling it again after the main loop (then the function would still have to handle '\0' as the next character of the text). The handling of '\0' in the inner loop is not that easy to get right; I got it working right with '.' only on the second attempt.

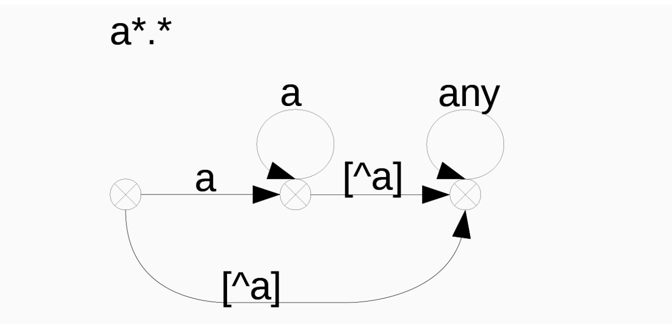


Figure 5: The improved FSM from Figure 4

The Value in Being Systematic

This is a good place to talk about how to fix a bug after it has been found. I've seen the people that are strong on intuition but not systematic start semi-randomly changing the spots that look vaguely plausible. I've literally seen a candidate do three wrong changes in a row, hoping every time that the issue will get resolved. This is the situation where thinking things through systematically really shines. Good questions to start with are, what do these values mean and how did their handling in the code diverge from this meaning? And then you can proceed to "Where did it happen?" and fix the bug. The same candidate, after I asked these questions, was able to find and resolve the bug on the first attempt in just a few seconds.

Returning to this solution, the problem that it solves is underspecified. It doesn't tell you what to do in case the pattern is invalid, either starting with a star or containing multiple stars in a row. This is by design, to see if the candidate will notice this and ask for a clarification, and my answer to this clarification question is, "What do you think is reasonable?" to see if the candidate is able to enumerate the pros and cons of different approaches: either return some error indication or handle it silently in some reasonable way.

I've made this solution do the silent handling, simply because it's easier to do in a small code snippet: it treats the "wrong" stars as literals. From the caller's standpoint it might be either good or bad: the good is that the caller won't have to handle the errors, and the bad is that the author of the incorrect pattern might be surprised by its effect and might never find out that it's incorrect.

But even this version is not great. The nested loops and re-parsing the pattern on each text character are convoluted; I got it right only on the second attempt. When the going gets hard, it's usually a good indication that a different approach should be tried.

What should the other approach be? It's up to your intuition to supply the ideas, for that's its line of work. This is why you need both intuition and systematics; one is not enough.

For this problem, it's much easier to go the other way around, iterating through the pattern and consuming the matching characters from the text:

```
bool match(const char *pattern, const char *text) {
    const char *t = text;
    for (const char *p = pattern; *p != 0; p++) {
        if (p[1] == '*') {
            if (*p == '.') {
                while (*t)
                    ++t;
            } else {
                while (*t == *p)
                    ++t;
            }
            ++p; // adjust to consume the star
        } else if (*p == '.') {
            if (*t++ == 0)
                return false;
        } else {
            if (*t++ != *p)
                return false;
        }
    }
    return *t == 0;
}
```

This version is much smaller and much easier to follow through. It explicitly selects by the type of each pattern element, so each one of them has its own code fragment, which avoids spreading its logic through the code and mixing it with the logic of the other elements. And all this makes the creation of bugs more difficult.

This whole problem is not very imaginative and can be solved well by just hammering out the code systematically. But this nice, short version contains an item that requires at least a little leap of intuition: it looks ahead by two characters, not just one, to detect whether the current pattern character is followed by a star. It's not something that's usually taught, but it makes the code a lot easier. As I like to say, it's not people for the programming patterns, it's programming patterns for the people. Don't be afraid to step away from a taught pattern if it makes your code better.

This version also has a theoretical foundation: it's a recursive-descent LL(1) parser of the text, except that the regular expressions define a non-recursive language, so there is no recursion. It really is perfectly per textbook; you've just got to pick the right textbook! It also parses, not a fixed grammar, but one given in the regular expression pattern. So it's an LL(2) parser of the pattern, with the nested LL(1) parsers of the matching substrings in the text. The 2 in LL(2) means that we're looking ahead by two characters. The pattern can also be parsed by an LL(1) parser, but looking ahead by two characters makes it easier.

Conclusion

This is the version that came to mind almost right away when I first thought about this problem. But I can't really say that it just popped into my mind out of nowhere. I do size up the different approaches in my mind intuitively and try the ones that look simpler first. It doesn't mean that this first estimation is always right. Sometimes I go pretty deep with one approach before deciding to abandon it and apply the lessons learned to another approach. And sometimes this other approach ends up being even worse, but the lessons learned there help to get through the logjam of the first approach.

So if you start with poor approaches, you can still arrive at better ones by listening to the hints that the code gives to you as you write it. When you see an easier way to go, use it. You can also power through the difficult approaches systematically to the successful end, but that tends to be much more difficult than switching the approach to an easier one. Intuition and systematic logic working hand-in-hand can get you much farther than either one of them alone.

Save the Date!

28TH USENIX SECURITY SYMPOSIUM

August 14–16, 2019 • Santa Clara, CA, USA

The 28th USENIX Security Symposium brings together researchers, practitioners, system administrators, system programmers, and others to share and explore the latest advances in the security and privacy of computer systems and networks.

The Symposium will span three days, with a technical program including refereed papers, invited talks, posters, panel discussions, and Birds-of-a-Feather sessions. Co-located workshops will precede the Symposium on August 12 and 13.

Program Co-Chairs
Nadia Heninger, *University of Pennsylvania*
Patrick Traynor, *University of Florida*

Registration will open in May 2019.

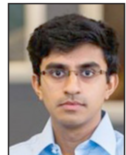
www.usenix.org/sec19

From Data Science to Production ML

Introducing USENIX OpML

NISHA TALAGALA, BHARATH RAMSUNDAR, AND
SWAMINATHAN SUNDARARAMAN

Nisha Talagala is co-founder, CTO/VP of Engineering at ParallelM, a startup focused on production machine learning. Nisha has more than 15 years of expertise in software, distributed systems, machine learning, persistent memory, and flash. Nisha earned her PhD at UC Berkeley on distributed systems research. Nisha holds 63 patents in distributed systems, algorithms, networking, memory architecture, and performance. Nisha is a frequent speaker at both industry and academic conferences and serves on multiple technical conference steering and program committees. She is the Program co-chair for OpML '19.
nisha@gprof.com



Bharath Ramsundar did his PhD in computer science at Stanford University where he studied the application of deep-learning to problems in drug discovery. While there, he created the deepchem.io open-source drug discovery project and the moleculenet.ai benchmark suite. Bharath is the co-author of *TensorFlow for Deep Learning: From Linear Regression to Reinforcement Learning* and the forthcoming *Deep Learning for the Life Sciences* with O'Reilly Media. As a co-founder of Computable, Bharath is focused on designing the decentralized protocols that will unlock data and AI to create the next stage of the Internet. bharath.ramsundar@gmail.com

Swaminathan (Swami) Sundararaman is the Lead Architect of ParallelM, an early stage startup focused on production machine learning and deep learning. Swami was previously at Fusion-io, Inc. and Sandisk Corp. He holds a PhD from the University of Wisconsin-Madison.
swaminathan.sundararaman@gmail.com

In this article we explain the challenges with deploying ML/DL models in production and how USENIX OpML can help bring participants for different disciplines to address the herculean task of safely managing the model life cycle in production.

Machine learning (ML) and its variants such as deep learning (DL) and reinforcement learning are starting to impact every commercial industry. The 2019 USENIX Conference on Operational Machine Learning (OpML '19), dedicated to operational machine learning and its variants, will focus on the full life cycle of deploying and managing ML into production. The goal of the conference is to help develop robust practices for scaling the management of models (i.e., artifact of learning from big data) throughout their life cycle. Through such practices, we can help organizations transition from manually hand-holding to automated management of ML models in production (i.e., ML version of the move in server operations from "pets to cattle" [9]).

Having engaged with hundreds of data scientists over the past few years, it was clear to us that while generating machine-learning models has become easier, moving them into production still remains challenging. It made us carefully think about the question, what is making machine learning more accessible on the one hand, but challenging for broad deployment on the other?

ML technologies have been around for many decades, with intermittent spikes of activity and interest. In the last few years, however, ML and DL technologies have been proven to work effectively in real world use cases in many domains. This shift is driven by several factors:

- **The Data**: Devices from sensors to robots are generating increasing amounts of rich data (from simple value time series to images, sound, and video). While the data itself is valuable, its ultimate benefit to a business's bottom line comes from the analytics that extract the insights hidden within. While simple data sets (such as streams of individual values) can be analyzed via database queries or complex event-processing techniques, the increasing richness of data (multiple correlated mixed type streams, images, sound, video) requires more complex ML and DL approaches. The increased volumes of data also enable ML/DL algorithms to achieve peak efficiency.
- **The Compute**: The ubiquity of high performance commodity computing, driven by both massive core count increases in individual CPUs and low-cost cloud computing services, have made it possible to match data growth with similarly scalable ML and DL capabilities. Hardware innovations such as GPUs, custom FPGAs, and instruction-set support in modern CPUs have further improved ML algorithm performance, making it practical to train using massive data sets [1].
- **The Algorithms**: The availability of open source algorithms for ML and DL via libraries for analytic engines like Spark, TensorFlow, Caffe, NumPy, scikit-learn [2], just to name a few, now offers a massive range of algorithmic techniques for the data scientist sandbox. With open source, even the most state-of-the-art algorithms in research are frequently publicly available to test, tune, and use, nearly as soon as they are invented.

These trends addressed the first issues impeding real-world ML (the data, the compute, and quality algorithmic implementations). The next problem was finding a data scientist to match the specific business problem and data set to a suitable algorithm. A lot has been written about the shortage of data scientists [3]. This issue, while real, has been actively addressed in the last several years with online data science courses, specialty programs in universities for data science, and tools that simplify model creation (the democratization of data science) [5]. The latest approach to mitigating this problem, AutoML [4], promises to automate the process of model creation and selection, making it even easier to improve the productivity of a single data scientist.

These trends have also helped generate lots of models. However, to be useful for any application, the model has to be deployed in production with its outputs (recommendations, classifications, etc.) connected to the application that needs it. Deploying, managing, and optimizing ML/DL in production incurs additional challenges:

- **Real-World Dynamism**: Depending on use case, incoming data feeds can change dramatically, possibly beyond what was evaluated in the data scientist sandbox. This in turn affects production ML behavior in ways that are hard to predict or detect via standard production means.
- **Expertise Mismatch**: On one side, IT operations administrators are experts in deployment and management of software and services in production. On the other side, data scientists are experts in the algorithms and associated mathematics. Operating ML/DL in production requires the combined skills of both groups.
- **Non-Intuitive Complexity**: In contrast to other intuitive analytics like rule-based, relational database or pattern matching key-value-based systems (where the output can be predicted from the input values), the core of ML/DL algorithms are mathematical functions (i.e., models) whose data-dependent behavior is not intuitive to most humans.
- **Reproducibility and Diagnostics Challenges**: Since ML/DL algorithms can be probabilistic in nature, there is no consistently "correct" result. For example, even for the same data input, many different outputs are possible depending on what recent training occurred and other factors (such as parameters used to train a model).
- **Inherent Heterogeneity**: Many classes of ML algorithms exist (e.g., machine learning, deep learning, reinforcement learning), and specialized analytic engines (Spark, TensorFlow, PyTorch, containers to train/serve models via Kubernetes) have emerged, each excelling at some subset [2]. Practical ML solutions frequently combine different algorithmic techniques, requiring the production deployment to leverage multiple engines. This makes the deployment process even more fragile than the current data ingestion and processing pipelines. This is uncommon in other application spaces. In databases, for example, standardizing on a single type of DB for a workflow can be a useful production norm.

The term *Cambrian explosion* has already been used in several contexts to describe the growth of AI [6, 7]. Within this trend, what we are seeing now is the explosion of models in the data scientist sandbox, models that cannot be practically used until they are able to deliver on their promise in production. As the number of data scientists increases, as democratization and AutoML tools improve data science productivity, and as compute power grows making it easier to test new algorithms in sandbox, more and more models will be developed, each one awaiting the move into production use.

To help meet this challenge and support the growing community of ML researchers and engineers, data scientists, IT and DevOps engineers who are working to manage ML in production, several of us in industry have worked with USENIX to launch the first conference dedicated to Operational Machine Learning (OpML).

The goal of this conference is to bring the research and industry technical communities together to develop and bring to practice impactful research advances and cutting edge solutions to this problem. Unlike existing conferences and workshops, OpML will focus on "the final stage of deploying and managing ML into production and the subsequent continuous ML/DL lifecycle in production." This covers deployment, automation, orchestration, monitoring, diagnostics, compliance, governance, and the challenges of safely operating and optimizing production systems running ML/DL/Advanced algorithms on live data.

OpML will also provide several benefits for industry and academic participants (please see CFP for details in [8]). Submissions were due on February 15, 2019.

We invite you to participate in the inaugural OpML conference that will be held on May 20, 2019, in Santa Clara, CA, USA.

References

[1] "Nvidia Morphs from Graphics and Gaming to AI and Deep Learning," ZDNet, September 8, 2017: https://www.zdnet.com/article/nvidia-morphs-from-graphics-and-gaming-to-ai-and-deep-learning/.

[2] M. Heller, "Review: The Best Frameworks for Machine Learning and Deep Learning," InfoWorld, February 1, 2017: https://www.infoworld.com/article/3163525/analytics/review-the-best-frameworks-for-machine-learning-and-deep-learning.html.

[3] V. Zhang and C. Neimeth, "Three Reasons Why Data Scientist Remains the Top Job in America," InfoWorld, April 14, 2017: https://www.infoworld.com/article/3190008/big-data/3-reasons-why-data-scientist-remains-the-top-job-in-america.html.

[4] AutoML: http://www.ml4aad.org/automl/.

[5] M. Dillon, "The Democratization of Data Science and the Emergence of Citizen Data Scientists," *Daily Californian,* May 26, 2017: http://www.dailycal.org/2017/05/26/democratization-data-science-emergence-citizen-scientists/.

[6] G. Leopold, "Nvidia CEO Predicts AI 'Cambrian Explosion,'" HPC Wire, May 25, 2017: https://www.hpcwire.com/2017/05/25/nvidia-ceo-predicts-ai-cambrian-explosion/.

[7] S. Condon, "Google's Fei-Fei Li: Vision Is AI's 'Killer App,'" ZDNet, May 19, 2017: https://www.zdnet.com/article/googles-fei-fei-li-vision-is-ais-killer-app/.

[8] USENIX OpML Call for Participation: https://www.usenix.org/sites/default/files/opml19_cfp_121319.pdf.

[9] R. Bias, "The History of Pets vs Cattle and How to Use the Analogy Properly," September 29, 2016: http://cloudscaling.com/blog/cloud-computing/the-history-of-pets-vs-cattle/.

Save the Date!

Fifteenth Symposium on Usable Privacy and Security

Co-located with USENIX Security '19
August 11–13, 2019 • Santa Clara, CA, USA

The Fifteenth Symposium on Usable Privacy and Security (SOUPS 2019) will bring together an interdisciplinary group of researchers and practitioners in human computer interaction, security, and privacy. The program will feature technical papers, including replication papers and systematization of knowledge papers, workshops and tutorials, a poster session, and lightning talks.

Registration will open in May 2019.

Symposium Organizers

General Chair
Heather Richter Lipford, *University of North Carolina at Charlotte*

Technical Papers Co-Chairs
Michelle Mazurek, *University of Maryland*
Rob Reeder, *Google*

www.usenix.org/soups2019

COLUMNS

Python

PETER NORTON

Peter works on automating cloud environments. He loves using Python to solve problems. He has contributed to books on Linux and Python, helped with the New York Linux Users Group, and helped to organize past DevOpsDays NYC events. In addition to Python, Peter is slowly improving his knowledge of Rust, Clojure, and maybe other fun things. Even though he is a native New Yorker, he is currently living in and working from home in the northeast of Brazil. pcnorton@rbox.co.

This column is being written in December, which ends another year, which brings end of year holiday plans, deadlines (like the one for this column), and a chance to challenge yourself to do something different before the year is fully out and done.

For my part, over the last few years I've had a great time participating in the annual Advent of Code (adventofcode.com), which is a great way to take a break from work where you have to solve problems on a deadline, and… well, solve problems on a different deadline. But for fun.

It's a great opportunity to learn more about your preferred language, to try out a new language, or revisit how things work in a language you haven't used in a while. You can also compare notes with others and see how different languages can give you the tools you need (or how hard it is to build them from scratch if that's more to your taste).

For anyone who hasn't participated in one of these online advent calendars, this one involves creating a puzzle around Santa, elves, and a story arc adventure that you are on that gets Santa closer to delivering presents for all the good girls and boys. Each day you get a story, a problem description, an example of the data and what the results will be of the problem being presented (yes, tests), and a data set that's created for you so that your answer shouldn't work for anyone else (though the solutions should, of course). The answers are usually an integer, summing up all the work you've done. And you're rate limited to one answer per minute, so you can't just brute force the answer.

The problems are very much programming puzzler/interview type questions designed to let you stretch your computer science legs—data structures, complexity, etc. without having any serious stakes—and if you complete the puzzle you move on; it's all just for a good time. The problems are introduced day-by-day, but if you haven't done the challenge already, you can always visit the site as you read this column and participate if you feel like it.

What I enjoy about this is that it is so well executed. Very few programming interviews that I've seen are as well thought out as the Advent of Code, which speaks volumes for the organizers. The organizers have some themes—a variety of problems that require some knowledge that may be common in some jobs and problem domains but which in others can be novel and outside of the comfort zone.

Big-O Traps

One of the things you notice quickly is that solving the problems naively will lead you to quadratic solutions that will take forever with the size of the input you're provided. So one of the fun parts is getting to think about each particular problem, to think about the big-O characteristics of your code, and realizing your input is large enough to cause your computer to spin and struggle uselessly until the heat death of the universe.

These problems often run over familiar themes—some will involve iterating over lists, finding your way around other data structures forward and backward, over and over. As you may imagine, if you start with a little bit of bookkeeping, that sometimes turns into a lot of bookkeeping, which is a lot of hassle. When that starts to happen, it's helpful to step back. When

you can, sometimes stepping back includes treating the data like streams. In Python this basically means iterators and generators are your friends who take away the tedium. What's interesting and disappointing about this great and fun approach is how as you get more sophisticated with using iterators, you can sometimes get subtle and surprising behaviors, which aren't particularly well-documented (at least as far as I've seen).

Iterator Side Effects

With all that said, this year's Advent of Code had me encounter one of these side effects, one that I found quite surprising. It is simple, but I do think that in real-world usage it would cause hard-to-find bugs.

The specific behavior is in the `zip()` built-in function. If you've never used it before, it's sometimes easier to think of as syntactic sugar sprinkled over having to assign multiple variables in a loop. It can turn the following somewhat tedious code:

```
def odious(l1, l2, l3, l4, l5):
    """each argument is a list"""
    min_len = min(map(len, (l1, l2, l3, l4, l5,)))
    for iteration in range(min_len):
        v1 = l1[iteration]
        v2 = l2[iteration]
        v3 = l3[iteration]
        v4 = l4[iteration]
        v5 = l5[iteration]
        print(f"{v1}, {v2}, {v3}, {v4}, {v5}")
```

into something much simpler. This prints each element of the lists in the arguments as a group—first, all of the first elements, then all of the second elements, etc. The short, `zip()`-ified way of doing this looks like:

```
def melodious(l1, l2, l3, l4, l5):
    for v1, v2, v3, v4, v5 in zip(l1, l2, l3, l4, l5):
        print(f"{v1}, {v2}, {v3}, {v4}, {v5}")
```

Which is still clear and easy to understand. Since zip can work with any number of iterables, it's pretty flexible. It's been in Python since 2.0, and there's a lot more to read about it in PEP 201 at https://www.python.org/dev/peps/pep-0201/.

I also found the behavior of iterators interesting. Iterators are thoroughly ingrained in Python and feel very natural to use. However, they have a very specific definition, and if you want to know exactly what that is, I encourage you to read PEP 234: https://www.python.org/dev/peps/pep-0234/.

As I mentioned above, iterators allow us as Python programmers to have a potentially lazy stream of items, with only a few tradeoffs. On the upside, you can have infinite input that you can iterate over easily with `for` or `next()`; you can compose them with comprehensions and with really cool functions available in the `itertools` module! And iterators have led to generators with `yield` and generator comprehensions. A lot has been written in these pages about iterators, generators, co-routines, etc., so I will refer anyone interested to the excellent material in past *;login:* issues, which have gone into a lot of depth and breadth on the matter.

The downside of the tradeoff for how excellent iterators are is that we lose some of the flexibility of having a list or a special type or class whose position and indexability puts it entirely under our control. For an iterator to be useful, we must know that we're going to use it from beginning to end in a linear fashion—no rewinding, arbitrary glances at indexes, etc. In so many cases this is not a limitation but is specifically and exactly what we want, which is why iterators are so fantastic.

So, with that said, let me talk about the interesting problem that I ran into. The code involved looks something like this (in Python 3.7):

```
import itertools
def walk_forward(char_iter):
    """Consume input_iter, which is an iterator that provides
    one character at a time. When two characters match the
    filter criteria, remove them both and break so that the
    data can be walked backward to see if the new state has
    affected the keep_list.

    returns a list of characters that we want to keep
    """
    first_char = next(char_iter)
    keep_list = list()
    for second_char in char_iter:
        result = keep_or_remove(first_char, second_char)
        if not result:
            # Don't put the result into the keep list
            return keep_list
        keep_list.append(first_char)
        first_char = second_char
    return keep_list

def walk_backward(keep_list, char_iter):
    """A match has been found, and now we want to know if the
    combination of the last letter in the keep list, and the
    first letter in the char_iter could start eliminating each
    other. Essentially this works from the middle out as long
    as the characters would be eliminated. Once we find a pair
    that are keepers, we can exit from here and resume walking
    forward.

    Returns a list of characters - those that we still want to
    keep.
    """
    #Walk the keep_list backward
    first_gen = (x for x in keep_list[-1::-1])
```

```
    for first, second in zip(first_gen, char_iter):
        result = keep_or_remove(first, second)
        # Return the results in the same order we got them
        if result:
            return list(itertools.chain([second, first], \
             first_gen])[-1::-1]
```

This works fine—with a `main()` function that walks forward until there is some elimination, then walks backward, then forward, and so on. This should basically work to eliminate pairs of letters that match the `keep_or_remove()` function, which I haven't included here.

The hidden problem in `walk_backward` is that the use of `zip` will always try to consume the first element from each iterator. So when the `keep_list` is shorter than the remaining contents of `char_iter` (as it is likely to be towards the beginning), everything is fine. However, if it's the second iterator that becomes exhausted, as may happen, then `zip` will have already consumed from the `first_gen`, and you can't put it back. So, in this case, you may have lost data. It's only one datapoint, which is exactly enough to make people very upset in the right circumstances, that is, outside the world of fun puzzles.

Now that we've looked at this with some more context, let's look at a simpler reproducer case:

```
>>> a = (x for x in 'abcde')
>>> for first, second in zip(a, ()):
...   print(f"{first}, {second}")
...
>>> rest = list(a)
>>> print(f"{rest}")
['b', 'c', 'd', 'e']
```

Working Around the Problem

Once I understood the issue, it bothered me because working around it made the program harder to read since the obvious workaround is tedious. Tedious solutions beg for better ones, especially when they're for fun. However, in this case it also led me to wonder why there isn't already a better solution, and maybe a bit about whether my idea of a better solution was in fact better at all.

If this were a problem that a lot of people cared about, a PEP on it would probably have appeared. I expect that since this is a small wart in one tiny part of the language, most people with work to do would solve this by avoiding `zip`, or by not using iterators, relying instead on lists or similar types with known, queryable lengths and ensuring that these lengths were uniform for each argument to `zip`, which is the sweet spot for a safe and reliable `zip`. This thought makes me sad because it would be nice if Python offered a better way to handle this.

So let's think about it a bit more and see what comes out of it.

One simple approach to fixing this problem would be to make a more robust iterator, and doing that is pretty easy. However, to be useful it would require the iterator protocol to be more robust. For example, you could envision a new class that allows some interrogation, like peeking or, maybe a bit less ambitious, the ability to ask whether it's primed (by which I mean it still may have more values in the future) or stopped (`StopIteration` has been raised) without losing a value.

Unfortunately, these aren't small self-contained decisions. A fundamental thing like altering the behavior of the iterator protocol would probably, in the worst case, mean that every battery-included function or expression that consumes an iterator and handles `StopIteration` would have to know that there is this new capability, which is now a lot of work with a lot of sharp edges ready to poke you.

So let's just start with the easy part for now, and we can explore the harder parts later.

Taking advantage of the iterator protocol, let's start with a naive first try—we'll write an iterator that lets us ask whether there's more data while otherwise behaving like a regular iterator.

```
class SnitchIterator(object):
    def __next__(self):
        while True:
            return next(self.iterator)

    def __iter__(self):
        return self

    def __init__(self, src):
        """Using a source iterator, list, etc. create a new
        iterator that lets you non-destructively ask if there
        is a next element or not"""
        self.iterator = iter(src)
    def more(self):
        try:
            res = next(self)
            if res:
                self.iterator = itertools.chain([res], \
                 self.iterator)
                return True
        except StopIteration:
            return False
```

Now we can ask "Is there more to this?" and get an answer. But to solve the earlier problem, we'll also need a slightly different `zip` function to take advantage of this new feature, or else we're at a dead end. The special-case `zip`, or `snitch_zip`, would look like this:

```
def snitch_zip(*args):
    """Iterables must be a container, not an iterator. We must
    be able to go through them more than one time"""
    if False in ['__iter__' in dir(it) for it in args]:
        raise TypeError('All variables in *args must have \
         __iter__')
    while True:
        for series in args:
            if not series.more():
                raise StopIteration
        yield [next(series) for series in args]
```

You can see that creating a modified `zip` is pretty easy. However, this becomes a special case, which detracts from the simplicity of the iterator model, is going to perform worse than the built-in `zip`, and will probably have issues that we will cut ourselves on. There's nothing wrong with doing this for yourself when the use is appropriate, but it feels like something that, to be useful, would be better if it were in the language or at least in the standard library.

Doing something like this in the core language might have some niche usefulness but would come with the potential to break a lot of existing code, or at least make that code confusing. Some languages have macros and other practices to enable extending existing functionality for experimentation, and Python has at least one project that does this as well. If I can, I'll see if I can get `zip` to work with the `SnitchIterator` and discuss that next time.

Governance Follow-Up

Also, as a follow-up to the last column, the vote for the new governance model for Python has been counted, and PEP 8016, the steering council model, has been accepted: https://www.python.org/dev/peps/pep-8016/.

This means that the BDFL model will be replaced by a five-person elected steering committee with the goal of taking care of the language, and they will be subject to oversight by the core team members—those who actively contribute to the community.

You can see the results of the actual vote at https://discuss.python.org/t/python-governance-vote-december-2018-results/546.

Again, I encourage anyone interested to follow this process closely.

Happy New Year!

Practical Perl Tools

So Long and Thanks for All the Fish

DAVID N. BLANK-EDELMAN

David has over 30 years of experience in the systems administration/DevOps/SRE field in large multiplatform environments. He is the curator/editor of the O'Reilly Book *Seeking SRE: Conversations on Running Production Systems at Scale* and author of the O'Reilly Otter Book (*Automating Systems Administration with Perl*). He is a co-founder of the wildly popular SREcon conferences hosted globally by USENIX. David currently works for Microsoft as a senior cloud advocate focusing on site reliability engineering.

After 12 continuous years of writing this column with only one missed month, it is time for this column to shuffle off this mortal coil and leave room in *;login:* for a different column.

I am so, so grateful to:

- You, the reader. It's been a thrill to be able to talk with you each issue about something interesting in the land of Perl.
- USENIX, who gave me the challenge to stretch myself each issue to find that interesting topic.
- The countless authors and contributors in the Perl world that I've had the pleasure of writing about.

You may (or may not) be wondering: just how many Practical Perl Tools columns have been published in that 12-year span? I know I was. I thought it might be fitting to show you one last Perl program that I wrote to help me find all of the previous columns and also answer this question. Ready for one last dance?

For this code, we return to an old friend that has appeared in this column before, `WWW::Mechanize`. This module makes it easy to fetch web pages and parse them for specific links. The first part of the code sets up where we are going to pull the information from and grabs the first page.

```
use strict;
use WWW::Mechanize;
use open qw(:std :utf8); # quash warnings due to UTF-8 chars

# where are the issues found?
my $start = 'https://www.usenix.org/publications/login';

# for finding my articles
my $name  = 'blank-edelman|practical-perl-tools';

my $mech  = WWW::Mechanize->new;

# fetch the issues page
$mech->get( $start );
```

That page is both a listing of all of the issues and the root for all of the subsequent pages we will want to fetch. In the code we're going to see, we are careful to only retrieve URLs that start with this prefix.

Now let's find all of the issues we will want to check for an article:

```
my @issues = $mech->find_all_links(
 tag => "a",
 url_abs_regex =>
   qr/$start\/[a-z-]+20(0[6-9]|1[0-8])/,
   text_regex => qr/.+/,
);
```

The `find_all_links()` method is doing all of the heavy lifting, but we should explain the arguments it is receiving. The “tag” argument is pretty easy to guess: we’re only looking for the anchor HTML tag, things of the form `<a href='something'>text</a>`. The next two arguments are a little more obtuse.

The first, `url_abs_regex`, is a regular expression meant to only find certain links on the page. It serves two purposes in this case: only select links that begin with `$start`, and also limit which years will be selected. I happen to know I began writing the column in 2006, so it only finds 2006–2009 and 2010–2018.

The `text_regex` deals with a quirk in the source of the issues page. Each issue actually has two anchor tabs, one for the picture of the cover, the second is the link for the text name (e.g., “Summer”). This regex makes sure we only grab one of the two, the one that has any characters in the text portion of the URL. This means we choose:

Spring

instead of:

The end result of the call to `find_all_links` is a list of `WWW::Mechanize::Link` objects that will point to all of the possible issues we’ll want to scan for this column.

Now let’s iterate over all of the issue links we found:

```
my $issue_count = 0;
foreach my $issue (@issues){

       $mech->get($issue->url_abs());

       my $article_link = $mech->find_link(
               url_regex=>qr/$name/,
       );

       if (defined $article_link){
               print  $article_link->text() .
               ":\n" .
               $article_link->url_abs(),"\n\n";
          $issue_count++;
       }
}
print "$issue_count issues in total!\n";
```

For each issue link we have, we fetch the contents of that link, then look for links in that page which could be my column. If we find one, we print the name of the column and its URL.

It would be pretty simple to grab the actual PDF of the column at this point if we wanted to create an archive of the content. This would consist of another `get()`, `find_link()` to locate the PDF on the page, `get()` that URL, and finally a call to `save_content()` to write it to a file. Permit me one last “exercise for the reader” if you will.

The output of our code looks like this:

```
Practical Perl Tools: Top of the Charts:
https://www.usenix.org/publications/login/spring2018
/blank-edelman

Practical Perl Tools: It's a Relationship Thing:
https://www.usenix.org/publications/login/summer2018
/blank-edelman

Practical Perl Tools: GraphQL Is Pretty Good Anyway:
https://www.usenix.org/publications/login/fall-2018
-vol-43-no-2/blank-edelman

Practical Perl Tools: Off the Charts:
https://www.usenix.org/publications/login/spring2017
/practical-perl-tools-charts

Practical Perl Tools: Perl on a Plane:
https://www.usenix.org/publications/login/summer2017
/blank-edelman
...
66 issues in total!
```

And there’s the answer. Thank you, dear reader, for being with me for 66 columns.

Take care.

Executing Other Programs in Go

CHRIS (MAC) MCENIRY

Chris (Mac) McEniry is a practicing sysadmin responsible for running a large e-commerce and gaming service. He's been working and developing in an operational capacity for 15 years. In his free time, he builds tools and thinks about efficiency. cmceniry@mit.edu

If you have come to the Go world from bash or another shell language, one of the most critical tasks that you will be trying to replicate is calling out to other programs. Go has mechanisms in the standard library to accomplish this—the `os/exec` library.

When running an external program, you have to decide how to interact with this. These interactions tend to fall into several patterns:

1. Fire and Wait: Run another program, send its output to the terminal, and wait for it to finish.
2. Fire and Forget: Run another program, send its output to the terminal, and do not wait for it.
3. Pipe In: Feed data into the program.
4. Check Out: Check the output or exit code of the program.
5. Replace: Perform some setup, and then replace the current process with the other program.
6. Interact: Start another program and interact back and forth with it.

Each of these patterns is a combination of:

1. What to do with input for the other program?
2. What to do with the other program's output?
3. Do we need to block until the other program is done or not?

In this article, we're going to examine each of these interactions in turn with a focus on which patterns they use.

Note: These examples are very UNIX and bash focused. As such, the examples will only work on limited environments.

The code for these examples can be found at https://github.com/cmceniry/login/ in the exec directory. Each example is its own appropriately named subdirectory so that it can be executed directly with `go run $EXAMPLE`.

Fire and Wait

This is the simplest interaction with another process. In this pattern, the input and output are of little concern, but we do want to wait until the other program is complete. Its profile looks like:

1. Input: supply none (attaches automatically to `/dev/null` or equivalent)
2. Output: provide back to the attached terminal
3. Block till completion: yes

We begin much like any other Go program—the package declaration, imports, and our main func: the main library to include here is the standard library's `os` and `os/exec` components.

firenwait.go: setup.

```
package main

import (
   "os"
   "os/exec"
)

func main() {
```

To begin with the meat of our program, we first invoke the `exec.Command` func. This accepts the invocation of the other program as arguments. Go performs standard `PATH` resolution to find the program by name, but in our case, we're going to invoke the `/bin/ls` command. In addition, we pass `exec.Command` any arguments. For this example, we just want to list out the current directory's outputs.

As a result, we receive back an `*exec.Cmd` struct which will handle all interactions with our called program.

firenwait.go: command.

```
   c := exec.Command("/bin/ls", ".")
```

Since we want to display the output of the `ls` command, we need to connect the output of that command with our display. This is done by associating the `Stdout` member of our `*exec.Cmd` with the main `Stdout` from our current program. The main `Stdout` is available from the main `os` package.

Note: `Stdout`, and its accompanying `Stderr` for error output, is an `io.Writer` interface. Input is covered under `Stdin`, which is an `io.Reader` interface. If they are not specified by setting `Stdout` or `Stdin`, they default to `nil` and will be connected to the `/dev/null` equivalent. We'll explore using other items that satisfy the `Reader/Writer` interfaces later.

firenwait.go: connectoutput.

```
   c.Stdout = os.Stdout
```

With all of the initialization complete, we can `Run` our program. `Run` will block until the child process completes or fails. It returns an error if it is unable to run the other program or if the other program fails during execution (gets a non-zero exit code). For the example case, we `panic` for that, or exit normally otherwise.

firenwait.go: run.

```
   err := c.Run()
   if err != nil {
      panic(err)
   }
}
```

We can now run our example with `go run` and see the current directory. In this example, we are using `$GOPATH/src/github.com/cmceniry/login` as our starting point.

```
$ go run exec/firenwait/firenwait.go
README.md   exec      gofs      hardcode    useldap
```

Fire and Forget

The second example handles the case where we run a program but do not check for what happens to it. This follows the patterns for:

1. Input: supply none
2. Output: provide back to terminal
3. Block till completion: no

This is very similar to the first example. It includes the same libraries—plus `time` for the example. It creates the command the same way, and it associates the output in the same way. There are only two primary differences.

The first is the specific start of the command -- `c.Start()` instead of `c.Run()`. `Start` will begin the other process but will return as soon as it begins instead of waiting for it to complete. If there's an issue starting the other process—e.g., command is not found—then it will show up as the returned error to `Start`.

firenforget.go: start.

```
   err := c.Start()
```

The second is to reap the child when it exits. Although we're not doing anything with the output, we still need to handle the child when it exits. Otherwise, the child can hang around as a zombie process. It's not complete fire and forget—only mostly fire and forget.

firenforget.go: wait.

```
   go func() {
      err := c.Wait()
      if err != nil {
         panic(err)
      }
   }()
```

The last part is that we hold our program from finishing up for a couple of seconds. We want to make sure that our program exits after the other program exits. In most cases, there would be some other work that would be going on, so we simulate that with just a simple `Sleep`:

firenforget.go: work.

```
   // Do some other work...
   time.Sleep(2 * time.Second)
```

Pipe In

Our next example shows how to provide input to a program. As mentioned in the first example, `Stdin` is an `io.Reader`, so anything that satisfies that interface will work. In this example, we'll use the patterns from our first example—only "Input" is different:

1. Input: supplied
2. Output: provide back to terminal
3. Block till completion: yes

The goal of this example is to have the calculating program `dc` perform some arithmetic for us. We'll be using a `strings.Reader` to provide `dc` with data. With the following input, `dc` will calculate the sum of 1 plus 2, print the output, and quit.

```
1
2
+
p
q
```

The initialization is the same as previous programs, except for the addition of the `strings` package from the standard library.

As with the previous examples, we begin with getting an `exec.Cmd` struct. In this case, we invoke the `dc` command and supply no arguments.

pipein.go: command.

```
c := exec.Command("/usr/bin/dc")
```

Next, we connect the inputs and outputs. `strings.Reader` implements the `io.Reader` interface, so we can use it to send a static string in as our input. We connect this with the `Stdin` of our command. As before, we connect `Stdout` of our command with the existing terminal `Stdout`.

pipein.go: io.

```
c.Stdin = strings.NewReader("1\n2\n+\np\nq\n")
c.Stdout = os.Stdout
```

And now we can run `dc`.

pipein.go: run.

```
err := c.Run()
```

If all works out, we will see the sum as the result:

```
$ go run exec/pipein/pipein.go
3
```

Check Out

Normally, just running a command and expecting it to behave is wishful thinking. We can get some information if there's an issue starting the command, or with `Run` we can see whether the program exited with a non-zero exit code. However, sometimes it's important to know what that return code is or what the program returns as output.

In those cases, we need to check the `ProcessState` after our command runs. `ProcessState` is a very generic `struct` which mainly indicates whether the process is still running or not. For detailed information, it has a `Sys()` member method that returns an empty interface whose concrete implementation is very much operating system dependent. On UNIX, `Sys()` returns a `syscall.WaitStatus` that includes the detailed exit code that we're looking for.

In this example, we're going to run a command and check its exit code. It follows the pattern of:

1. Input: supply none
2. Output: discard except for the exit code
3. Block till completion: yes

The initialization is the same except that, in this case, we must include the `syscall` package of the standard library. We are even calling the command in the same way.

checkout.go: command.

```
c := exec.Command("/usr/bin/false")

err := c.Run()
```

Since we expect the failure to return an error, we must handle it. We check to see whether it is of the `exec.ExitError` type and handle that separately. Otherwise, we will panic on any other error, since that indicates something really unexpected happened, or exit normally on no error.

checkout.go: result.

```
switch err.(type) {
case *exec.ExitError:
   ws := c.ProcessState.Sys().(syscall.WaitStatus)
   fmt.Printf("Exited %d\n", ws.ExitStatus())
case nil:
   fmt.Printf("Exited normally\n")
default:
   panic(err)
}
```

If all goes well, we can see the expected result of an exit code of 1:

```
$ go run exec/checkout/checkout.go
Exited 1
```

You can see alternate behaviors by changing the command to execute. Try:

- /usr/bin/true
- /usr/bin/notfound

Replace

In the Replace interaction, we are largely using the Go program as a wrapper. The wrapper will perform some setup and then transfer control over to another program. Some examples of useful setups:

- Set environment variables—configuration parameters
- Set up file-system structures—working directory, lock files, etc.
- Check other dependencies—backend database or service—before starting up the application process

This follows the patterns:

1. Input: handed off
2. Output: handed off
3. Block till completion: no, handed off

Since process replacement is extremely operating system dependent, we're going to use the `syscall` package in the standard library—same as the previous example. This makes the program setup match the last exercise.

From there, we need to make any modifications as part of our wrapping action. In this example, we'll add a single environment variable.

replace.go: env.

```
env := append(
    os.Environ(),
    "USENIXLOGIN=true",
)
```

From there, instead of using the higher level `os/exec` package, we use the `syscall.Exec` function directly. For this example, we want to spawn a shell with the manipulated environment.

replace.go: handoff.

```
syscall.Exec("/bin/bash", []string{}, env)
```

For wrappers as simple as environment manipulations, that is the extent of it. We can now use the updated environment.

```
$ echo $USENIXLOGIN

$ go run exec/replace/replace.go
bash$ echo $USENIXLOGIN
true
```

Interact

The last example that we're going to take a look at involves interacting with the other program. This can be used if you need to programmatically interact with other command-line or terminal-based tools. Typically, you will be looking for data or errors and responding back into them.

Since this is before the process has exited, we're going to focus our time on manipulating the input and output of the process.

Specifically, in this example, we're going to:

- start with the letter "a",
- feed it into `cat`,
- read the output `cat` back out,
- append "b" to the output,
- feed that back into the same `cat` process, and
- repeat for "c", "d", and "e".

Each time through, we're going to build on the letters that have already been supplied, unless we're finally presented with the full string "abcde".

So far, we've been working with the `io.Reader` and `io.Writer` interfaces of `Stdin` and `Stdout`. To be able to provide the continuous feeds, Go provides a way to get pipes for each of these: `(*Cmd) StdinPipe()` and `(*Cmd) StdoutPipe()`. We're going to use these in this example to aid us.

For the start of our main section, we need to initialize our data and our command.

interact.go: vars.

```
feed := []string{"a", "b", "c", "d", "e", ""}
c := exec.Command("/bin/cat")
```

After that, we grab the pipes for `Stdin` and `Stdout`.

interact.go: stdin,stdout.

```
cin, err := c.StdinPipe()
cout, err := c.StdoutPipe()
```

We're going to rely on the `bufio` package of the standard library to more easily support the line and string manipulation that works well with `cat`. To do so, we need to wrap our `io.Reader` and `io.Writer` with `bufio.Scanner` and `bufio.Writer`, respectively.

interact.go: buffer.

```
bin := bufio.NewWriter(cin)
bout := bufio.NewScanner(cout)
```

With all of the prep work out of the way, we can get the ball rolling with `cat`. To do so, we need to prime the input with a newline and start `cat`.

interact.go: prime.

```
bin.WriteString("\n")
bin.Flush()
c.Start()
```

Next, we're going to iterate through our data. For each piece, we want to gather the `cat` output and then write back the output with our addition.

interact.go: addnprint.

```
for _, addition := range feed {
   if !bout.Scan() {
      panic("ended early")
   }
   if bout.Text() != "" {
      fmt.Printf("%s\n", bout.Text())
   }
   bin.WriteString(bout.Text() + addition + "\n")
   bin.Flush()
}
```

At the end, we want to clean up. Much like with the Fire and Forget example, we still need to wait for the other process to finish. However, since `cat` will not finish until its input is finished, we must first close that.

interact.go: cleanup.

```
cin.Close()
c.Wait()
```

Now, we can run our program much like the others, and we should see our five-letter output:

```
$ go run exec/interact/interact.go
a
ab
abc
abcd
abcde
```

Conclusion

One of the most basic functions of any script is to build on other programs. It is crucial to be able to both trigger other programs with various inputs and to respond to the results of those other programs. Although the invocation of these other programs has a few more steps in Go versus traditional scripting languages, Go allows you to more readily tap into a large corpus of software for processing inputs and outputs.

I hope this article has given you confidence to use Go when it is appropriate to handle these process interactions, and some ideas for how to readily do so.

USENIX Supporters

USENIX Patrons
Bloomberg • Facebook • Google • Microsoft • NetApp

USENIX Benefactors
Amazon • Oracle • Two Sigma • VMware

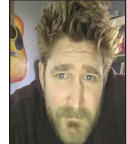
USENIX Partners
BestVPN.com • Cisco Meraki • Teradactyl • TheBestVPN.com

Open Access Publishing Partner
PeerJ

iVoyeur

Flow 3

DAVE JOSEPHSEN



Dave Josephsen is a book author, code developer, and monitoring expert who works for Sparkpost. His continuing mission: to help engineers worldwide close the feedback loop.
dave-usenix@skeptech.org

If you take any sort of guided tour of Paris, you are likely to hear references to "The Great Flood of 1910," wherein the Seine rose to a depth of eight meters above its normal height, buried the city in water, and shut down critical infrastructure like freshwater and heating-oil delivery for a month.

Rivers have backed up and flooded cities since time out of mind, but this flood makes for particularly great data-engineering metaphor fodder because the water never managed to overflow the tops of the quay walls lining the river itself. In other words, primary queue cardinality was within threshold.

Instead, the city was flooded from below by way of the recently enlarged and fortified sewer system that ran from every direction into the Seine. I suppose you could say that the hotpath bypassed the queue. Ironically, the infrastructure most prized by city planners, like train stations and hospitals, which had the best-engineered sewer access, were hit the worst. Their basement grates spewed water like the geysers of Yellowstone, rapidly flooding and spilling into the streets until the streets themselves became waterways.

In some areas of the city, firefighters used boats to rescue stranded people from second-story windows, as engineers constructed a city-wide series of wooden catwalks to enable residents to reach shelters and sources of food and fresh water.

Here's the thing: if you've never read anything about the history of Paris, the city was supposedly an untenable *mess*, until Napoleon III put it into the hands of a gentleman named Georges-Eugène Haussmann. "Baron Haussmann" would spend 20 years becoming the most unpopular guy in France as he demolished the medieval firetrap the city had been in order to singlehandedly re-architect it into the city we more or less recognize as Paris today.

The "grand rearchitecture" of the city included a herculean refactoring of the dense labyrinth of pipes, sewers, and tunnels beneath the streets into the most modern and robust sewer system in the world. The system provided the city's freshwater supply, steam heat, and oil pipes to power the streetlights, as it simultaneously swept away rainwater and waste. The sewers were such a source of pride that bureaucrats of the time used their own pet euphemisms to make them sound less like sewers and more like re-election.

Haussmann himself compared them to bodily organs. "The underground galleries," he said, "are an organ of the great city, functioning like an organ of the human body, without seeing the light of day; clean and fresh water, light and heat circulate like the various fluids whose movement and maintenance serves the life of the body; the secretions are taken away mysteriously and don't disturb the good functioning of the city and without spoiling its beautiful exterior."

It's fortunate Haussmann died before his miraculous "underground galleries" buried the city chest-deep in human waste and river water. Had he been there to see it, I'm sure it would have been the facepalm heard around the world.

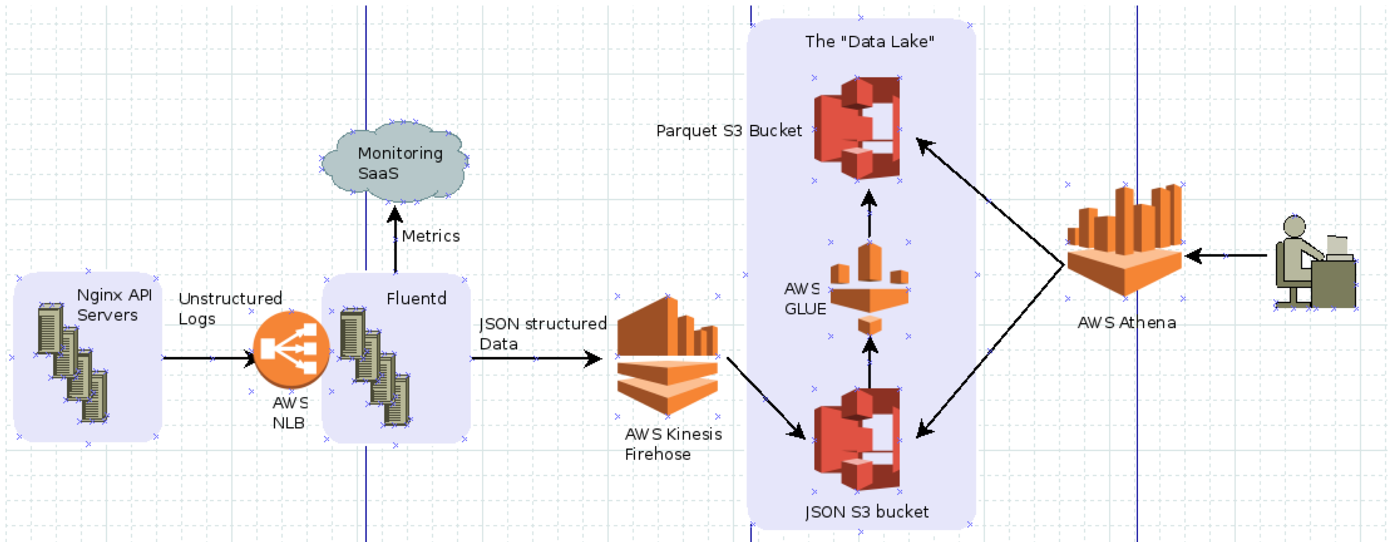


Figure 1: Sparkpost's "Internal Event Hose" data pipeline

I suspect that anyone who has seriously worked with data pipelines or distributed systems can probably relate; an overabundance of input can have extreme and unforeseen effects on asynchronous processing systems.

The Flow, Part Three

This is the third article in my series about our API-query data pipeline, so you, dear reader, could certainly be forgiven not knowing just what the heck I'm going on about. Let's pause, therefore, for a moment of reflection. In Figure 1 you can see the pipeline in its entirety.

In my last article, we spoke about the first data transformation, which takes place inside Fluentd, to change raw log data into structured JSON. We learned about how tags and message routing works inside Fluentd and about Fluentd's buffered output plugins. I also mentioned that we were using the Prometheus plugin to extract some metrics from Fluentd and shared some cardinality graphs from our production monitoring system, Circonus.

Merely enabling Prometheus in your `tdagent.conf`, along with its `outputmonitor` plugin, gives you all the visibility you need to detect backups inside Fluentd of the sort tour guides in Paris are *still* talking about a century later.

```
<source>
  @type prometheus
</source>
<source>
  @type prometheus_output_monitor
</source>
```

Upon restarting `td-agent` (the Fluentd demon), a `wget http://localhost:24231/metrics` will yield myriad stats on every registered output plugin, like these two counters of messages emitted per output plugin (`sns` and `firehose` for us):

```
fluentd_output_status_emit_count{plugin_id="object:3f86dc5b
80cc",type="amazon_sns"} 570277.0
fluentd_output_status_emit_count{plugin_id="object:3f86d983
3444",type="kinesis_firehose"} 10109263509
```

Fluentd also has a filter type Prometheus plugin, which you can use in your routing configuration to extract metrics directly from the data as it passes through. We use this to break down the cardinality of the various types of API calls that are occurring within our Nginx data. Here's the configuration blurb:

```
<filter firehose_parsed.**>
  @type prometheus
  <metric>
    name outgoing_msg
    type counter
    desc Outgoing messages
  </metric>
  <labels>
    type ${type}
  </labels>
</filter>
```

This filter catches all messages tagged with "firehose_parsed" and increments a counter metric named "outgoing_msg" that—crucially—is labeled with the *value* of the message's `type` attribute. In other words, as each message is routed through this filter, Fluentd literally uses the value of `msg.type` to create the

Prometheus metric label. Hence, when we `wget` the reporting socket, we get output metrics that break down the cardinality of each type of API call our customers are currently making:

```
...
outgoing_msg{type="get_sending-domains"} 31190473.0
outgoing_msg{type="get_subaccounts"} 33089429.0
outgoing_msg{type="get_webhooks"} 527765630.0
outgoing_msg{type="auth_request"} 58139133.0
outgoing_msg{type="get_users"} 144456173.0
outgoing_msg{type="4xx_error"} 193923362.0
...
```

In a proper Prometheus shop, we'd be using the Prometheus server to slurp up all of these metrics and report on them, but for better or worse, our monitoring solution of choice lies in another direction, so I wrote a small shell script that performs the polling and reformatting. Omitting the error handling, it's really just two lines...

```
INPUT=$(curl -k -ss -m "${TIMEOUT}" "${URL}")

echo "${INPUT}" | grep -v '^#' | sed -e 's/{.*="/`/' -e 's/"} //'
-e 's/ //g' -e 's/^fluentd_//' -e 's/`\([^`]\+\)$/ n \1/'
```

If you squint at it hard enough you'll see it transforms the output into *backtick* separated lines of the style: `outgoing_msg`get_sending-domains`31190473.0`. I know. Backtick separation. Don't get me started.

When we first architected this data pipeline we carefully read up on the various AWS streaming event services, compared their limits and tradeoffs against our workload, and decided that SNS was the best fit for us. We installed the most popular version of the SNS Fluentd plugin, gave it our configuration particulars, and watched everything collapse and fail in a Parisian-esque epic flood of traffic.

We eventually discovered two overlapping problems. The first, which I mentioned in my last article, was the SNS Fluentd plugin we found didn't support buffered output, meaning, among other bad things, that it didn't support threading and completely blocked the entire Fluentd process as it tried to flush 11,000 messages to SNS every second.

The second problem was that the SNS service itself doesn't have a bulk-send endpoint, so every message emitted equates to a single HTTP connection. It's surprisingly easy for little details like this to be obscured by frameworks and plugins and abstraction. Engineers who know AWS very well are fond of saying things like *there are no limits to SNS*, and asking around, I heard myriad utopian tales of shops pushing hundreds of thousands of 140-byte messages per second into SNS without breaking a sweat.

Well, it turns out, the real-world limit on SNS is the number of HTTP connections you can reliably make per second from your sending instance's ENI. I'm not really sure what that number is (it no doubt varies by instance type), but I'm here to tell you, for us, it was smaller than 11,000 divided by three instances.

Rather than attempting to scale up or out, we took a look at AWS Kinesis Firehose, which has a bulk-send endpoint capable of ingesting batches of over 100 messages in a single HTTP call. This was a WAY more efficient and reliable means of feeding data into AWS. Bonus, the Fluentd Kinesis plugin is well supported, buffered, and supports threading.

We experimented with lambdas attached to our firehose to transform the JSON log data directly in to Parquet but eventually decided that we wanted a copy of the data in both JSON and Parquet, so we pointed the firehose directly at an S3 bucket. Kinesis automatically partitions this data up for us into minute-sized chunks, ready for Athena to parse through them.

To make the final hop into columnar data format, we rely on a combination of custom-written code, Apache Spark, and AWS Glue. Spark's PySpark (http://spark.apache.org/docs/latest/api/python/index.html) library makes it simple to `sqlContext.read.json()` our JSON data from S3 into a Spark DataFrame (https://spark.apache.org/docs/latest/sql-programming-guide.html), and from there `df.write.parquet()` it back out to a new S3 bucket in Parquet format. We use AWS Glue to schedule our PySpark code as an ETL job that runs hourly (five minutes after the hour, to give firehose a sufficient buffer of time).

I find it difficult to articulate the extent to which this data has enriched my life as an engineer, but I'll give you an example from last week, wherein someone noticed that we appeared to be bouncing an order of magnitude more email than normal, which everyone found...worrisome.

I first checked whether there was a pattern of increased bounces for our top-tier receivers. This sort of thing has happened in the past when Gmail, for example, implemented some new, aggressive, and ill-conceived filtering technology.

```
select dt, count_if(routing_domain='gmail.com') as google,
count_if(routing_domain='yahoo.com') as yahoo,
count_if(routing_domain='hotmail.com') as hotmail
from "glue-data-lake-usw2-prd".eventlog_parquet
WHERE bounce_class=21
AND dt >= '2018-09-01'
group by 1;
```

With this Athena query, I was able to get a day-by-day breakdown since September 1 of email we bounced to the top three providers and verify that we were *NOT* in fact bouncing more mail than normal. This query took three minutes to complete and scanned around 100 GB of data (Athena queries cost $5 per TB scanned).

What, then, could account for the increase in bounce traffic?

```
select count(dt),raw_reason
from "glue-data-lake-usw2-prd".eventlog_parquet
WHERE bounce_class=21 and dt between '2018-10-01' and
'2018-10-21'
group by raw_reason
order by count(td)
LIMIT 10;
```

```
select count(dt),raw_reason
from "glue-data-lake-usw2-prd".eventlog_parquet
WHERE bounce_class=21 and dt > '2018-10-21'
group by raw_reason
order by count(td)
LIMIT 10;
```

With these two queries I was able to enumerate the top 10 reasons that email bounced in the period before the change was noted, and then again in the period after the change was noted. I discovered that there was indeed a difference between these two lists. The first looked like:

```
454 4.4.4 [internal] no MX or A for domain
554 5.4.4 [internal] Domain Lookup Failed
"550-Requested action not taken: mailbox unavailable
550 invalid DNS MX or A/AAAA resource record"
451 Your domain is not configured to use this MX host.
```

While the second looked like:

```
454 4.4.4 [internal] no MX or A for domain
554 5.4.7 [internal] message timeout (exceeded max time, last
        transfail: 454 4.4.4 [internal] no MX or A for domain)
554 5.4.4 [internal] Domain Lookup Failed
554 5.4.7 [internal] exceeded max time without delivery
```

As you can see, some new, timeout-related error messages have overtaken the first and fourth most common error message in the logs. As it turns out, our engineering teams had implemented a new suite of error detection code and had miss-classified these timeout messages as bounce-class messages, which in turn caused a reporting error.

While this particular example turned out to be a false-alarm rather than a flood, I think it serves to illustrate how capable our new log data pipeline is at helping us deal with the deluge.

I think that pretty much wraps up my series on our Data Pipeline at Sparkpost, and along with it, my overspilling (sorry) of river-related metaphor. Until next time.

For Good Measure

Patent Activity as a Measure of Cybersecurity Innovation

DAN GEER AND SCOTT GUTHERY

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

Scott Guthery holds over 50 patents for his work in cybersecurity. He worked for Bell Laboratories, Schlumberger, and Microsoft and co-founded Mobile-Mind with Mary Cronin to build secure mobile applications for the GSM SIM. He currently runs Docent Press, which publishes books on the history of mathematics, computing, and technology. sbg@acw.com

Type "cybersecurity" into Google Patents, sort by oldest and then newest, and take the top 100 in each list. Keeping in mind that the lists include applications as well as grants, Table 1 lists the number of entries by country in the respective lists.

Country	Oldest	Newest
Belgium	5	
Canada	4	
China	7	76
Denmark	1	
EU/WTO	8	1
Finland	2	
France	8	
Germany	9	
Great Britain	7	3
Japan	4	2
Korea	1	
Netherlands	2	
Spain	4	
United States	38	18

Table 1: Country sources are consolidating geographically

The top three assignees in the oldest list were AT&T/Bell Labs, Computer Security Corporation, and Westinghouse Electric in that order. The top three assignees in the newest list were two Chinese companies, and then IBM.

But what, you might ask, does this have to do with computer security metrics?

If you come up with a new and improved espresso machine and you wish to derive the maximum economic benefit from your invention, the two most frequently used methods of protecting your newly hatched intellectual property are applying for a patent or treating what is "new, useful, and non-obvious" in your espresso machine as a trade secret. If Table 1 were about espresso machines, the difference between the oldest and newest columns could reasonably be attributed to more companies selecting trade secret protection rather than applying for a patent.

That explanation is not as compelling for cybersecurity. A trade secret is "not generally known or reasonably ascertainable by others," but while it is possible that an innovation in cybersecurity is intended for use only within a (trade) secret context, this is not the typical business case. (This may well be the typical case in governmental and military contexts.) Because of the computer security community's aversion to secret sauce, if the inventor wishes to offer the invention in the cybersecurity marketplace, maintaining the protection of a trade secret becomes problematic; an enterprise you'd like to convince to license your innovation will want to know how it works, so protection leans more toward applying for a patent than toward using a trade secret as it would for that espresso machine. If you are going to be forced to reveal the inner workings of the invention in patent application detail, then you need to apply for a patent.

But still you ask, what does this have to do with computer security metrics?

Bruce Schneier is quoted on the Wikipedia page about elliptic curve cryptography patents ("ECC Patents") as saying in 2007, "Certicom certainly can claim ownership of ECC. The algorithm was developed and patented by the company's founders, and the patents are well written and strong. I don't like it, but they can claim ownership." Other companies hold patents on various cryptographic algorithms; the RSA patents come easily to mind.

More than a few standards discussions have wrestled with the inclusion of patented technology. Commercial entities holding a patent in such cases have every incentive to come to *fair, reasonable and non-discriminatory (FRAND) terms* for the use of their technology and thus for its use in a standard. Such was the case with both ECC and RSA. But this incentive is lacking when it is a governmental or regulatory entity that holds a patent. In this case the use of the patented technology can be required independent of any standards deliberations and in what may be very unFRANDly terms.

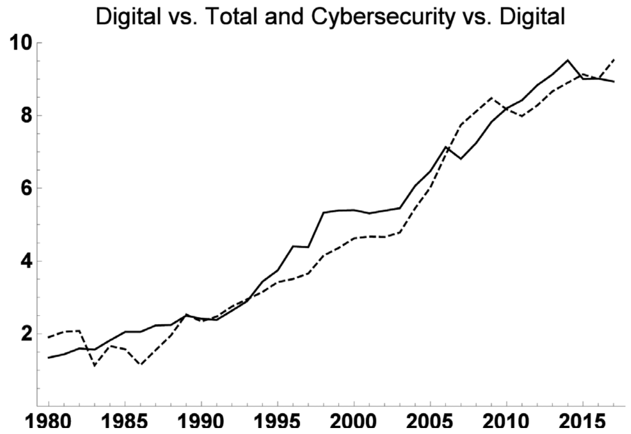


Figure 1: Digital patents as percentage of total number patents (solid) and cybersecurity patents as percentage of digital patents (dashed). All figures were drawn using data from http://www.patentsview.org/.

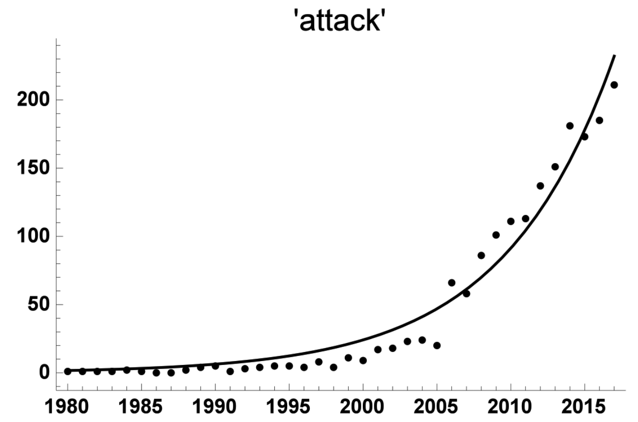


Figure 2a: Digital patents with "attack" in the patent abstract

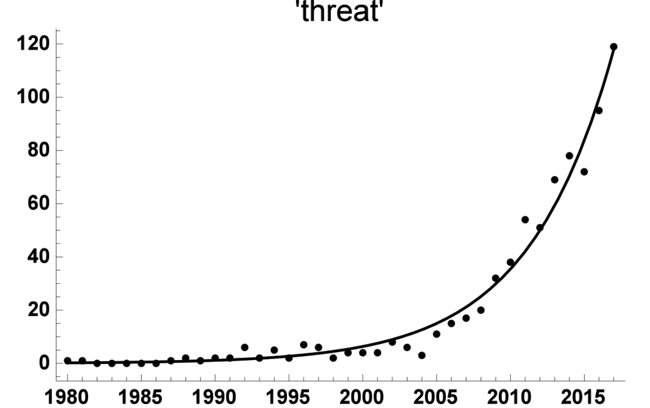


Figure 2b: Digital patents with "threat" in the patent abstract

"OK," you say, "I do care about the computer security landscape and who owns what plots of land, but this is more about the business of cybersecurity than about the bits and bytes. Do patent numbers have anything interesting to say here?"

Here are some words that appear in the titles of the patents in the newest list that don't appear in the titles in the old list (in alphabetical order):

> anti-theft, attack, authentication, detection, methods, threat, uncloneable

And here's the other way around, words in the old list titles that aren't in the new list titles:

> automatic, electric, filter, lock, switch, signals, transponder, telephone

Nothing certain can be deduced from this small sample, but one can glimpse a shift away from hardware toward protocols as well as a shift from offense toward defense.

Focusing now on granted US patents from 1980 to 2017 and, in particular, on the subset of these that have the word "computer" or "network" in the patent abstract, we will refer to this subset as digital patents. Within the set of digital patents, we will distinguish those whose abstract contains at least one of a list of cybersecurity words; we will refer to these as cybersecurity patents.

Figure 1 plots by year the ratio of the number of digital patents to the total number of patents issued (solid) and the ratio of the number of cybersecurity patents to the number of digital patents (dashed). One takeaway is that roughly speaking there is as much effort going into cybersecurity innovation within the domain of computers and networks as there is going into computers and networks overall.

Figures 2a and 2b plot by year the number of digital patents that have "attack" or "threat," respectively, in their abstract, together with an exponential fit to these counts.

If these plots were simply measuring the intensity of concern regarding attacks on and threats to computers and networks, then the exponential fits wouldn't be at all surprising. But they are measuring the number of "new, useful, and non-obvious" counters to attacks and threats which, in a world that might be thought of as settling into a day-in-and-day-out game of Spy vs. Spy, the exponentially growing number of pitches on which the game is being played might raise an eyebrow.

Posting a guard at the gate to check visitors' papers is a tried and true way of separating friend from foe. Figure 3a plots the number of appearances of "authentication" (upper/solid) and "credential" (lower/dashed) in the patent abstracts, while Figure 3b plots the number of appearances of "password" (upper/solid) and "biometric" (lower/dashed) in the patent abstracts.

Growth here is more linear than exponential of late, but the proliferation of new, useful, and non-obvious ideas is remarkable.

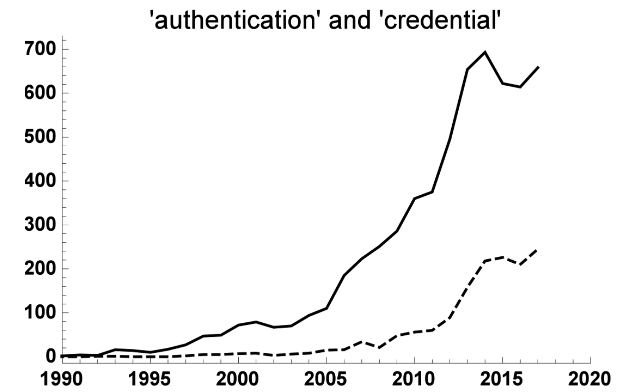


Figure 3a: Digital patents with "authentication" and "credential" in the patent abstract

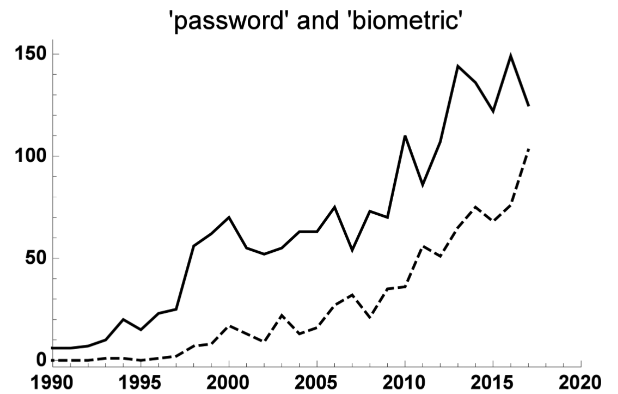


Figure 3b: Digital patents with "password" and "biometric" in the patent abstract

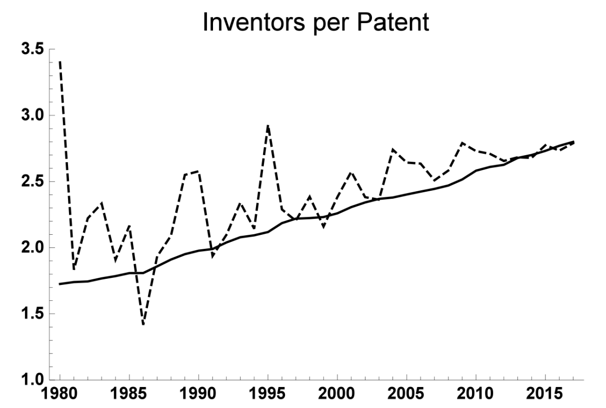


Figure 4: Inventors per patent: all patents (solid) and cybersecurity patents (dashed)

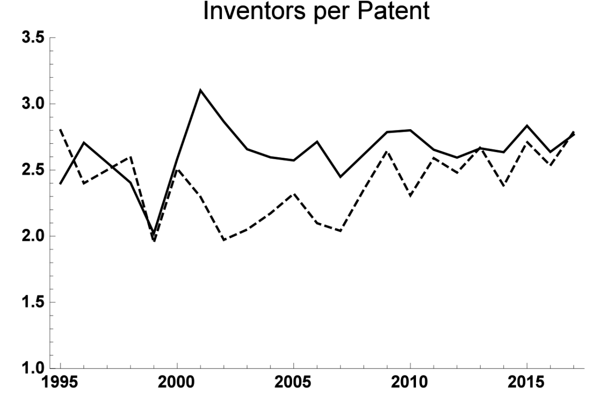


Figure 5: Inventors per patent: "authentication" (solid) and "biometric" (dashed)

Of course, that fact that the word "computer" appears in a patent abstract does not mean that the patent is about computers, and the same holds true for all of the other search terms discussed above. Nonetheless, one can safely conclude from this cursory analysis of the set of granted patents that inventive genius is ever harder at work on the cybersecurity problem.

Figure 4 shows the number of inventors per patent for all US patents and for cybersecurity patents. The fact that the number of inventors per patent has been growing slowly is well-known, and it comes as no surprise that whatever is driving this growth applies to cybersecurity patents *in toto* as well.

Curiously, if we restrict our attention to patents having to do with identity, the upward trend disappears. Figure 5 plots by year the average number of inventors for digital patents that have the word "authentication" (solid) or "biometric" (dashed) in their abstract. Roughly speaking, the average number of inventors per patent for patents having to do with identity is constant at about two and a half. Whatever it is driving the trend for most patents seems to be absent for this highly restricted subset.

The summary so far: where patent applications are coming from geographically has consolidated all but completely. Patents are probably the only strategy choice for cybersecurity inventors because users demand transparency in cybersecurity work much more than in other technical fields of endeavor. The subject-matter focus of cybersecurity patents may be moving toward defense (though it is possible that dual-use patents just avoid delineating their offensive capabilities). The fraction of all applications that are cybersecurity related is rising steeply, fueled by a growing fraction of all applications that are computer related and a growing fraction of computer-related applications that are for cybersecurity, growth compounded and compounded again. For any of these curves to radically change their course would surely mean something important.

We ask whether there really are this many new, useful, and non-obvious advances in cybersecurity. If there are, is this fast-rising tide of cybersecurity patents an unarguable confirmation of an equivalently fast-expanding digital attack surface? Or does the rising production of cybersecurity patents represent a correspondingly rising appreciation of the level of extant risk; that is to say, is society playing furious catch-up ball? Or is it something else again? Is it good or not good that while other sectors of the technological society require steadily larger and larger teams to come up with new, useful, and non-obvious ideas, in cybersecurity the teams are the same small size they have been for so long?

/dev/random
Ambush Computing

ROBERT G. FERRELL

Robert G. Ferrell, author of *The Tol Chronicles*, spends most of his time writing humor, fantasy, and science fiction. rgferrell@gmail.com

I recently read a piece about "ambient computing," which is the idea that the next logical phase of digital evolution is to have computers controlling our lives without the nuisance of us having to operate them. I would argue that this revolution has already taken its nascent steps. Even a fossilized semi-Luddite like me is a prisoner of the fitness app on my iPhone, for example. Gotta get those steps in—and Heaven forbid I should walk down the hall without said tracker and so fail to get "credit" for the trip. That app is like a jealous lover who insists on monitoring my every move.

I have studiously avoided the frankly terrifying specter of the personal digital assistant who listens to—and worse, tries to interpret—every sound I make. I cannot comprehend why anyone would voluntarily allow their residence to be bugged while at the same time paying for the privilege. These digital Mata Haris with cutesy monikers that sound like European commuter compacts or adult film personalities are nothing less than corporate surveillance technology of the highest order.

Do you really want the intimate details of your physiology and psychology reduced to mere datapoints for the sustenance of invisible robot overlords? I know I don't. Let us cast our nets out into the tepid waters of speculation and see what chimeras we drag in.

The odyssey begins as you're walking up your front steps. The doorbell camera has found a match for your facial features and welcomes you by name, while the porch mat measures your gait and mass. The latter data is transmitted to the nearest Bluetooth node, where it is made available to other household smart devices so that they may nag you about your ballooning weight and sell you plus-sized apparel at the same time.

In the foyer your smart coatrack reminds you that rain is forecast for later in the day. The first of several motion detectors scattered about the domicile records your location, direction of travel, and velocity, thereby predicting that you are heading for the kitchen and preparing it for your arrival. As you enter the room the dishwasher pops open with clean dishes, while the icemaker changes cube size and shape according to preferences that its predictive algorithm has deduced from prior encounters. The refrigerator, meanwhile, has rearranged the order of frozen entrées in its shelves based on time of day and your prior choices.

The microwave reads the UPC label of your entrée for heating times and ingredients. If it concludes the meal contains more sodium or sugar than is good for you, it will sound an alarm and even refuse to prepare it. If you employ manual override, it will start running ads for dieting products and heart-healthy vitamin supplements.

Your après manger bathroom visit includes a toothbrush that adjusts for tartar buildup and any detected gum disease, while the mirror checks for signs of acne, melanoma, eczema, receding hairline, and numerous other conditions, offering ads for remedies via text message. Your toilet has its own set of sensors, analyzing...well, what you would expect them to analyze. It might tell your intelligent pantry to increase the number of high-fiber meals it suggests, along with offering ads for same.

Your medicine cabinet has its own two cents to put in, of course. Not only does it keep track of prescription medications and submit refill requests on your behalf, it offers helpful tips on products that purport to control chronic conditions like headaches, muscle pain, dizzy spells, and foot odor—the presence of which it surmises by your pharmacological habits. If it perceives frequent visits to your primary care practitioner, it might tell your produce drawer to suggest daily apple consumption.

The home electronic entertainment naturally enumerates your behavior in excruciating detail. Every second you spend online or streaming to some device is examined, profiled, categorized, and exhaustively analyzed. The very best bargains in televisions and monitors can be had on models with integrated cameras and microphones, ostensibly for your own convenience in the quest to share absolutely every moment of your life with friends, family, and social media followers, but incidentally also to enable even more intrusive surveillance to benefit advertisers. Everybody wins, right?

Tying this tangled mass of data collection together is a central repository or database server. It doesn't have to be a single machine, though: your house is already quite likely a mesh network hosting its very own data cloud. Aren't you special? The Internet of Things quite definitely includes your Things. Conveniently, you don't even have to do anything to accomplish this web of integration. It just magically happens, like climate change and congressional oversight.

Let's not forget those little buttons all over the house that are supposed to order products for you at one touch. What timesavers they are! No more dreary comparison shopping or coupon clipping: simply trust that the button brings you what you need. So modern. So monolithic. So antitrustworthy.

Your doorbell, your thermostats, your security system, your kitchen appliances, your toiletries, your home entertainment system, your telecommunications devices…all of them working together to make your life—and the lives of those who want you to be unable to avoid their advertising at every turn—easier.

Since many of these doohickeys phone home on a regular basis, you can bet your data makes the same trip in first class accommodations. Once comfortably ensconced at the far end, it is packaged and sold to anyone who ponies up the requisite cash. The real beauty of this arrangement is that you actually pay for the equipment used to spy on you—sometimes even on a recurring basis if you're subscribed to a service connected to it.

Not that any of this is remotely novel or even recently invented: talking consumers into bankrolling their own exploitation is in fact a time-honored Madison Avenue tradition. Graduate theses have been written on the various techniques for achieving it. Careers have been built on it. Mansions, private jets, yachts, and even islands have been purchased from its proceeds. Taxes from those proceeds have been adroitly evaded. This, then, is the cycle of commerce.

I seem to have slid down a slippery slope from ambient computing to tax fraud, but in my defense, there weren't many obstructions. Happy fishbowl consumerism, y'all.

Book Reviews

RIK FARROW AND MARK LAMOURINE

The Site Reliability Workbook: Practical Ways to Implement SRE
Niall Murphy, David Rensin, Betsy Beyer, Kent Kawahara, and Stephen Thorne
O'Reilly Media, 2018, 512 pages
ISBN: 978-1-492-02950-2

Reviewed by Rik Farrow

When I think of a workbook, I expect something that contains exercises and complements an existing book or course. *The Site Reliability Workbook* fits the second part of that description. The authors intended that *TSRW* expand upon the best-selling *Site Reliability Engineering*, in part because of all the questions raised by readers of the first book.

Today, you can find all of the *SRE* book online, and as *TSRW* relies on that book, there are frequent references to chapters in the earlier book, all as bit.ly-shortened URLs. While that's useful, there are often summaries to the material, and I found that all I needed were the summaries to recall enough for the current material to make sense.

And instead of exercises, you get examples, case studies, and more in-depth descriptions. Right away I could see how useful this was in making the principles described in *SRE* concrete. There is even a chapter on Non-Abstract Large System Design, with tangible examples of what the authors, including Salim Virji, were teaching during LISA tutorials, a step-by-step approach to designing a reliable service for monitoring AdWords.

There was criticism that *SRE*, both the practice and the book, were something only Google, and a handful of companies like it, could take good advantage of. *TSRW* attempts to dispel those objections, largely by including authors outside of Google for many of the sections.

You will often find that books written by many authors have an uneven writing style. *TSRW* doesn't read that way at all: the writing remains clear, consistent, and easy-to-read throughout.

As to the argument that *SRE* is only for large organizations, I found myself thinking many times as I read *TSRW*, "If only I had known that 35 years ago." In the chapter about On-Call, I read about many practices that would have made my life easier in my first Bay Area job and prevented burnout. I also encountered some things I had tried to do, with partial success, in that long ago era. In other words, even if you don't consider yourself an SRE, there are definitely things you can learn from this book.

Managing Kubernetes: Operating Kubernetes Clusters in the Real World
Brendan Burns and Craig Tracey
O'Reilly Media, 2018, 188 pages
ISBN: 978-1-492-03391-2

Reviewed by Mark Lamourine

Often it seems that sysadmins are forgotten when people are writing documentation. It is common to see books for service users and for API developers. When it comes to managing services, it feels like the first response is to try to write some kind of GUI to smooth over the sharp bits and pretend they don't exist. This leaves the sysadmin needing to understand, manage, and diagnose complex systems with little guidance but their own wits and experience.

Managing Kubernetes won't solve every sysadmin problem, but it does go a long way toward illuminating the dark interior of one of the hottest buzzword services of the last few years.

Brendan Burns is one of the three original authors of Kubernetes and is still one of the top three contributors. With Craig Tracey, he provides the clearest description I've seen of the moving parts that, together, make a Kubernetes cluster.

Kubernetes is a distributed software container management service. That's quite a mouthful. If you're not already familiar with software containers, you should really start somewhere else. The most well-known container runtime system is Docker. There are others, but Docker is the BASIC programming language of containers. You'll be back quickly, because standalone containers have limited value. They come into their own when you start combining single-purpose containers into complex applications. How you combine them and then deploy them to make working services is what Kubernetes is all about.

Kubernetes is itself a (mostly) containerized service, built up of a number of cooperating service components. The hosts that participate in the clusters are called *nodes*. All nodes must have a container runtime environment such as Docker already installed and running.

Some nodes, called *head nodes*, are special. These run the management components and provide the brains of the cluster. The remainder of the nodes, called *worker nodes*, run components that control local containers and provide network communications. All of these coordinate by communicating with an API service that is distributed across the head nodes.

The arc of the book is a little different from most. Burns and Tracey don't have the reader attempt an installation until almost halfway through, in Chapter 6. Ordinary users would want to get started creating containers as soon as possible, but the sysadmin's purpose is to understand what is happening underneath when normal users start their work. The authors devote the first half of the book to describing the structure that installation will create.

In the second half of the book, the authors walk the reader through common operational processes. Many of these are concerned with providing and controlling access to the cluster. Users interact with the cluster by making requests to the API server. The next three chapters detail how user requests are validated and accepted.

The authors provide one of the better explanations I've read of the distinction between *authentication*, *authorization*, and what they call *admission*, which I might have called *policy*. In each case, they provide examples of the REST data structures that implement the communication protocol. The examples demonstrate the rationale and the structure, but none of them are meant to be comprehensive. The authors know that the Kubernetes project documentation [1] provides detailed specifications, though I do wish they had provided the appropriate links in-line with the text.

The final three chapters cover additional operational concerns: networking, monitoring, and disaster recovery. Again, the discussion is meant to give the reader a starting point for understanding what is possible and where to learn more. It is not a run-book but, rather, is concerned with architecture and taxonomy. It provides references to resources that the reader can use to learn and plan for a deployment.

Rather than being an operator's manual or a comprehensive reference, *Managing Kubernetes* describes the purpose and basic configuration of each component and gives the reader a sense of the structure and dynamics of Kubernetes as a whole. I have noted in other places that it is often very useful to understand any technology at least one layer, and preferably two, beneath the level where you mean to work. For both operators and architects of Kubernetes services, *Managing Kubernetes* will provide the peek beneath the covers.

Reference

[1] Kubernetes REST API specification: https://kubernetes.io/docs/concepts/overview/kubernetes-api/.

Learn Git in a Month of Lunches

Rick Umali
Manning Publications, 2015, 352 pages
ISBN: 978-1-617292415

Reviewed by Mark Lamourine

Many authors can't seem to decide whether they want to write a reference or a tutorial, often making their book less than ideal for either the beginner or the experienced reader. Rick Umali doesn't make this mistake. He knows he's writing a book for beginners, and *Learn Git in a Month of Lunches* is ideal for his audience.

Git is well suited to this kind of learning. It is a tool that is used by software developers to organize and manage their work. It allows them to share their work in a way that makes conflict avoidable or at least manageable. It has one purpose and a well-defined set of operations to accomplish that purpose. Tools like this are often learned fitfully, by experience, looking up the single solution to a single problem then going back to work. Umali has offered a straightforward and complete path for learning to use the most important capabilities of Git and the grounding to explore and learn more.

Umali, to his credit, dodges several common problems that arise from trying to present material in a narrative format. He avoids creating a contrived straw-man project. Instead, each chapter focuses on just one task or subcommand, and he discusses the most common aspects of that task. He does interlace examples for the three common platforms, Windows, Mac, and Linux, but each example is clearly labeled and distinguished by graphical conventions.

He also starts at the true learner's beginning (after installation) by creating an empty local repository. While most work in the real world will involve a remote repository, Umali leaves that for Chapter 12, well past the halfway point in the book. That first half is dedicated to getting comfortable with Git and just managing files in a repository. I was reminded firmly that all of the common operations, committing, cloning, branching, merging, and viewing logs are local operations. In every case the pattern for a file reference is first a local path that can then be extended to a URL by adding a standard prefix.

That said, the next four chapters cover the details of working with remote repositories; push, sync, rebase, and a chapter on branching conventions and collaborative workflows.

He wraps up with chapters on third-party Git software, working with GitHub, and configuration and tuning.

The “month of lunches” format limits the size of each chapter. This is a good thing. Umali crafts each one so that it is complete and self-contained. He encourages readers to spend a bounded time reading and then to go away and think and practice on their own. No chapter is longer than 20 pages. The longest ones are those with a lot of graphics. They either showcase the GUI interface or are concerned with the theory of revision control and so use lots of drawings to show the workflow for the reader. I’m not a good judge of GUI tools, but the base level introduction Umali offers is comparable to the CLI capabilities, and for those who like graphical tools it should serve well.

This is a beginner’s book, but I will pass it on with compliments. I did pick up a number of tips and ideas that will stick with me.

Gamestorming: A Playbook for Innovators, Rulebreakers, and Changemakers
Dave Gray, Sunni Brown, and James Macanufo
O’Reilly Media, 2010, 288 pages
ISBN 978-0596804176

Reviewed by Mark Lamourine

Brainstorming is a term in common use. To me it means going somewhere different (even if only in my head), preferably with a couple of my most trusted co-workers, presenting a problem I have in its broadest terms and then throwing around ideas without judgment or ego until something grabs all of our attention. Then we play with a couple of the “best” ideas until we better understand the problem, the challenges that remain, and, most importantly, what we want to try next. This is a very unstructured concept, and other people will have a different vision of what brainstorming is.

Gamestorming is a book that offers a lot of different ways to structure that communal thinking process.

The main idea of *Gamestorming* is to use the framework of a “game” to direct and focus the thinking and sharing process in a way that suits the particular goals of the session. A game, according to Gray, Brown, and Macanufo, is defined primarily by a play space, a set of rules, and a goal. With this loose but clear definition, they set out to give the reader a sense of how game play in a working context can lead both to the results that might elude more conventional planning sessions and to the relevant tools to get those results.

Chapters 2 and 3 present that toolbox. A moderator’s job in these kinds of planning meetings is to create an environment that will promote participation and cooperation. There are any number of ways the plan can be derailed. Chapter 2 enumerates 10 “essentials” that are the material needs for a good session. In Chapter 3 the authors lay out the skills and tactics that a moderator should have in order to be able to guide the participants and avoid ratholes and pitfalls.

The body of the book is four chapters that are a catalog of core games, those for opening, closing, and for exploring an idea space. The authors make a clear distinction between games meant to start a session and generate lots of wild ideas and those that are meant to refine and then focus on one concept and come to a close. In longer planning sessions the games might be chained together, or they can be played in separate sessions over a longer period of time if needed.

You may have visions of whiteboards and flip charts and multicolored sticky notes, and you wouldn’t be wrong. Most of us won’t use *Gamestorming* in day-to-day life as a software developer or sysadmin. The subtitle of the book, “A Playbook for Innovators, Rulebreakers, and Changemakers,” feels a bit grandiose to me. Many of the games are fairly common in dramatic training, especially those aimed at creating group coherence. I suspect very little here would be surprising to professional moderators or facilitators.

But we’re not that kind of professionals. I think, used judiciously, the ideas here could be helpful to those of us who find ourselves in that position despite our inclinations (or our best efforts). Sometimes it might be a good thing to shake us out of the stale format of our regular planning meetings, standups, or retrospectives. In that case, *Gamestorming* would be a good resource for getting ourselves into the mindset of a facilitator. For the hour or so it takes, perhaps a game is a good way to engage a whole team on a common problem and uncover a solution no one had thought of or felt invited to voice.

NOTES

USENIX Member Benefits

Members of the USENIX Association receive the following benefits:

Free subscription to *;login:*, the Association's quarterly magazine, featuring technical articles, tips and techniques, book reviews, and practical columns on such topics as security, site reliability engineering, Perl, and networks and operating systems

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

Registration discounts on standard technical sessions registration fees for selected USENIX-sponsored and co-sponsored events

The right to vote for board of director candidates as well as other matters affecting the Association

For more information regarding membership or benefits, please see www.usenix.org/membership/, or contact us via email (membership@usenix.org) or telephone (+1 510.528.8649).

USENIX Board of Directors

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

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

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

SECRETARY
Michael Bailey, *University of Illinois at Urbana-Champaign*
bailey@usenix.org

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

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

Kurt Andersen, *LinkedIn*
kurta@usenix.org

Angela Demke Brown, *University of Toronto*
angela@usenix.org

Amy Rich, *Nuna Inc.*
arr@usenix.org

EXECUTIVE DIRECTOR
Casey Henderson
casey@usenix.org

Notice of Annual Meeting

The USENIX Association's Annual Meeting with the membership and the Board of Directors will be held on the evening of Monday, July 8, in Renton, WA, during the week of the 2019 USENIX Annual Technical Conference.

Community Survey: Some Answers, Some More Questions

Liz Markel, Community Engagement Manager

In early fall, USENIX asked its community members for their opinions on a variety of topics through its Community Survey—our first survey of its kind since 2013. We solicited responses across a variety of media including our email newsletter; our social media channels including Facebook, Twitter and LinkedIn; the USENIX website and blog; and personal outreach. At the conclusion of the response period, more than 1,000 individuals had taken the time to share their thoughts, and we are incredibly appreciative of your participation in this process.

We expect the analysis to be ongoing throughout the first half of this year. As we mentioned in the opening of the survey, we are aiming to:

- Paint a data-driven picture of the USENIX community.
- Assess community members' perceptions of the organization.
- Evaluate membership options and determine what USENIX can do to better serve our communities.
- Gather information that will help USENIX make strategic decisions about various timely issues.

To address all of those priorities, we had to ask a lot of questions, including inquiring about demographics; more on that below. As an acknowledgement of the time commitment this survey required, we offered high-value raffle prizes to six randomly selected participants who completed the survey.

Going forward, our plan is to make this survey an annual opportunity to hear from you and to help guide important organizational

Women in Advanced Computing Birds-of-a-Feather session at LISA18.

LISA18 Program Co-Chairs Rikki Endsley and Brendan Gregg deliver their opening remarks.

LISA18 attendees spend some time chatting and connecting during a break.

The evening reception at LISA18 included the opportunity to screenprint your own shirt.

Denelle Dixon, Mozilla, delivers her Enigma 2019 talk, "It's Not 'Our' Data: Do We Want to Create a World of No Surprises?"

decisions. Did you participate in the survey? If not, why not? If there was something that prevented you from answering this year, please provide your feedback to liz@usenix.org! We will take into account your input for the 2019 edition of this survey. Additionally, if you have any other things on your mind, you can always share those with me as well.

Why Demographics?

Some respondents wanted to know why we cared about demographic data, especially questions inquiring about gender and ethnicity. Demographic data is a way to slice and dice responses to other questions that help us identify trends that may be related to community member attributes such as age, gender or ethnicity.

For example, the gender imbalance in USENIX's community is reflective of the gender imbalance in the computing systems community at large. As part of our mission, we're striving to mitigate this issue through offerings like the Women in Advanced Computing Birds-of-a-Feather sessions at our conferences, and Diversity Grants that offer funding for conference travel to many underrepresented groups in the field, including but not limited to women.

By filtering responses to non-demographic questions against the gender demographic question, we are able to identify specific needs from the women in our community, and build programs around those needs that are more likely to be successful because they are in direct response to identified needs. As we gather more survey data year over year, we can track the overall gender balance for USENIX's community and see if we are moving the needle in the right direction.

There were some other key questions that required demographic data, including the following:

Who is in our community now? How is our community changing over time?

We've discussed the gender question above a bit, but there are other defining elements of our communities that will directly impact how we put our mission into practice. For example, think about your career: with respect to networking, knowledge growth, and skills development, your needs have likely changed over time. Understanding the fields in which our community members work, the length of time they've spent in their areas of work, where they are on the spectrum of their career's lifetime, and whether or not they've pursued advanced degrees is important with respect to the content we produce for our conferences, as well as the additional support we provide for networking and professional advancement.

That evolution can also affect how we are communicating with you. Do you want more or less interaction on social media? How valuable is in-person communication for you? (Answer according to your responses: very valuable, and your responses indicate that this does not vary by age!)

Just as your individual career and your communications needs have evolved, so too have the needs and the face of the larger community. With regular surveys and year-over-year data, we can stay on top of these changes and adjust our programmatic offerings accordingly to be as supportive of you as possible.

Are we effectively serving those who are in our community?

What about those who might be part of our community in the future? We already have policies in place like our USENIX Conference Code of Conduct and Guidelines for Speakers that spell out our position on harassment (tl;dr: we don't tolerate it, and there is a reporting and enforcement process). What other policies are necessary for our current and future community members to ensure a positive experience for them while they are participating in USENIX-supported activities? With a demographic portrait of our community, we can continue to create and enforce relevant policies and support the growth of the advanced computer systems profession.

Where are you?

USENIX is an international organization, and we would like to continue to increase

our international presence via our conferences. We have primarily done this with our SREcon events to date, and community members are showing up! For example, more than 55% of all of the survey respondents who indicated they have attended SREcon Europe/Middle East/Africa said they reside in one of those regions. For survey respondents who have attended SREcon Asia/Australia, just over 30% identified as residents of the region. We are excited about the success of these events, and the local response, as well as the rich exchange of ideas that comes from folks visiting other parts of the world and finding out the issues that affect particular regions. Questions tied to survey participant geography will help us consider future conference locations, both domestic and international.

Who Are You?

So, who is the USENIX community comprised of? While we know that there are many communities underneath the umbrella of USENIX, we were curious to know how you defined those communities for yourselves.

When we designed the survey, we debated about how to ask which community you belong to: practitioner or academic? Sysadmin or SRE? We ended up with two questions: one that asked about conference attendance, and one that asked respondents to self-select their areas of work. Our expectation was that the responses to these questions would be consistent. We also had certain expectations about where overlaps of interest and work would occur.

Our very preliminary analysis of these responses was surprising. For example, many people who identified as LISA attendees and/or sysadmins also identified as USENIX Security attendees and/or those working in areas related to security, but this does not correlate with the profiles of those who have registered for USENIX Security, meaning that the two items should be mentioned distinctly. How does this overlap affect what is happening in industry and academia? Can USENIX facilitate productive collaboration in this area? What does this mean for our conference content?

We are asking some more questions of the data, but we also want to ask you: how do you define the professional community you are a part of? Do you consider yourself part of communities that your work supports? How important is engaging with that community, and how do you go about that engagement? How do you decide which conferences to attend? Please send me your thoughts: liz@usenix.org.

Food for Thought

Of course, one of the potential (and potentially fun) outcomes of doing research is that you wind up with more questions than answers. Many of the thoughtful responses provided throughout the survey have prompted other questions on the following topics.

Volunteering

The majority of my professional work for the past decade has been alongside committed and talented volunteers. When I joined the USENIX team, I was immediately impressed by the corps of volunteers involved in the organization whose subject matter expertise and leadership is a significant part of our success.

Conferences are a big part of who USENIX is and what we do. Many of you said you would be interested in helping at conferences, and I find myself wondering what new roles volunteers could fill that would enhance attendees' experiences—especially first-time attendees—and create a more fulfilling and valuable conference experience. For example, a conference I recently participated in as an attendee asked local attendees to volunteer to organize small dinners at nearby restaurants. It was an opportunity for new attendees to see a bit of the city and easily meet people in what might have otherwise been an overwhelming environment. Think back to your first time attending a particular conference: did you participate in an event like this? How did it help your overall event experience? If this wasn't an opportunity, did you wish

Enigma 2019 Program Co-Chair Franzi Roesner, Enigma Steering Committee member Parisa Tabriz, and USENIX Executive Director Casey Henderson enjoy one of the evening receptions in Burlingame.

Max Smeets of Stanford University delivers his Enigma 2019 talk, "Countering Adversarial Cyber Campaigns."

Nicholas Weaver of the International Computer Science Institute (ICSI) and University of California, Berkeley delivers his Enigma 2019 talk, "Cryptocurrency: Burn It with Fire."

Enigma 2019 Student and Diversity Grant Recipients

there was something like this to help you break the ice and make connections? Would you like to give back to our current conference participants and provide a meaningful experience? Again, I would love to hear from you with your thoughts on these questions, and encourage you to reach out to me via email, find me at a conference and share your feedback, or include your comments on a post-conference survey.

For this particular idea of attendee dinners to come to fruition, we would need a number of things to fall into place—including willing volunteers. While that particular idea is germinating, I'd ask you to consider what other ideas you have for volunteer-driven activities on-site at conferences that would improve the conference experience, and that you would be willing to lead or participate in. Make sure to tell me about them by sending me an email at liz@usenix.org.

Building My Reading List

If you send me your ideas related to conferences, I would love to hear about books, blogs, podcasts, e-newsletters, and other resources that I should know about, too! Many of you mentioned *Wired* magazine in your survey responses as one of the publications you frequently read. I've combed through the back issues of *Wired*, but I'm ready for more in the new year to help me better understand the work that you do. My goal is twofold: gain more insight into your work so that USENIX can serve you better, but also understand the relevance of your work to the general public, which will inform my conversations when advocating for USENIX outside of your community.

In the interest of fair exchange, if you send me your resource recommendations, I'll leverage my English degree and experience serving librarians and will send you some book recommendations sure to keep you entertained on flights to USENIX conferences. You might also check out the book reviews section of *;login:* for excellent suggestions, too!

How Are We Doing?

If you've read any of my previous USENIX Notes entries, you may have noticed my genuine enthusiasm for USENIX's work, and my belief that we're doing some really great work, both in terms of the content and conference experience we provide.

Respondents to our survey question about how we're doing on fulfillment of our mission tend to agree with my assessment: on a scale of 1 to 4, from (1) needs significant improvement to (4) amazing work, we earned the following weighted averages for each area of our mission:

- Foster technical excellence and innovation: 3.3
- Support and disseminate research with a practical bias: 3.3
- Provide a neutral forum for discussion of technical issues: 3.2
- Encourage computing outreach into the community at large: 3

This is a great starting point, but there is still room for improvement: our performance, our ability to meet your needs, our communication with you about what we're up to. It gives us a measuring stick as we consider where to put our resources and creative energy in the coming months.

Several open-ended comments from respondents have me thinking beyond these results: many spoke highly of what USENIX has to offer and the value we deliver. These same respondents also wondered why more people aren't aware of USENIX. The idea of USENIX as a "best kept secret in advanced computing systems" does have some allure, but we'll be a much greater force for good if more people know about our work and get involved. How can we accomplish this? How can you help?

My inbox is always open: liz@usenix.org.

Bob Lord presents his Enigma 2019 talk, "Mr. Lord Goes to Washington, or Applying Security outside the Tech World."

Daniela Seabra Oliveira delivers her Enigma 2019 talk, "Why Even Experienced and Highly Intelligent Developers Write Vulnerable Code and What We Should Do about It."

Two conferences' worth of Enigma leadership: Franzi Roesner, Ben Adida, and Daniela Seabra Oliveira.

SAVE THE DATES!

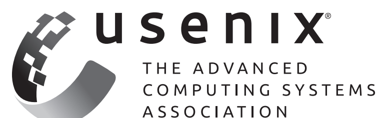
MARCH 25–27, 2019 • BROOKLYN, NY, USA
www.usenix.org/srecon19americas

JUNE 12–14, 2019 • SINGAPORE
www.usenix.org/srecon19asia

OCTOBER 2–4, 2019 • DUBLIN, IRELAND
www.usenix.org/srecon19europe

SREcon is a gathering of engineers who care deeply about site reliability, systems engineering, and working with complex distributed systems at scale. SREcon challenges both those new to the profession as well as those who have been involved in SRE or related endeavors for years. The conference culture is based upon respectful collaboration amongst all participants in the community through critical thought, deep technical insights, continuous improvement, and innovation.

Follow us at @SREcon



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

Save the Date!

USENIX ATC '19

2019 USENIX Annual Technical Conference

JULY 10–12, 2019 • Renton, WA, USA

USENIX ATC '19 will bring together leading systems researchers for cutting-edge systems research and the opportunity to gain insight into a wealth of must-know topics.

Program Co-Chairs:
Dahlia Malkhi, *VMware Research*, and Dan Tsafrir, *Technion—Israel Institute of Technology & VMware Research*

Co-located with USENIX ATC '19

HotStorage '19: 11th USENIX Workshop on Hot Topics in Storage and File Systems
July 8–9, 2019
www.usenix.org/hotstorage19

HotCloud '19: 11th USENIX Workshop on Hot Topics in Cloud Computing
July 8, 2019
www.usenix.org/hotcloud19

HotEdge '19: 2nd USENIX Workshop on Hot Topics in Edge Computing
July 9, 2019
www.usenix.org/hotedge19

Registration will open in May 2019.

www.usenix.org/atc19

usenix
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION