;login:

FALL 2016 VOL. 41, NO. 3

usenix
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

usenix
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

UPCOMING EVENTS

OSDI '16: 12th USENIX Symposium on Operating Systems Design and Implementation
Sponsored by USENIX in cooperation with ACM SIGOPS
November 2–4, 2016, Savannah, GA, USA
www.usenix.org/osdi16

Co-located with OSDI '16

INFLOW '16: 4th Workshop on Interactions of NVM/Flash with Operating Systems and Workloads
November 1, 2016
www.usenix.org/inflow16

LISA16
December 4–9, 2016, Boston, MA, USA
www.usenix.org/lisa16

Co-located with LISA16

SESA '16: 2016 USENIX Summit for Educators in System Administration
December 6, 2016
Submissions due September 19, 2016
www.usenix.org/sesa16

USENIX Journal of Education in System Administration (JESA)
Published in conjunction with SESA
www.usenix.org/jesa

Enigma 2017
January 30–February 1, 2017, Oakland, CA, USA
enigma.usenix.org

FAST '17: 15th USENIX Conference on File and Storage Technologies
Sponsored by USENIX in cooperation with ACM SIGOPS
February 27–March 2, 2017, Santa Clara, CA, USA
Submissions due September 27, 2016
www.usenix.org/fast17

SREcon17
March 13–14, 2017, San Francisco, CA, USA

NSDI '17: 14th USENIX Symposium on Networked Systems Design and Implementation
March 27–29, 2017, Boston, MA, USA
Paper titles and abstracts due September 14, 2016
www.usenix.org/nsdi17

SREcon17 Asia
May 22–24, 2017, Singapore

Do you know about the USENIX open access policy?

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

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

www.usenix.org/membership

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

;login:

FALL 2016 VOL. 41, NO. 3

usenix
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

EDITOR
Rik Farrow
rik@usenix.org

MANAGING EDITOR
Michele Nelson
michele@usenix.org

COPY EDITORS
Steve Gilmartin
Amber Ankerholz

PRODUCTION
Arnold Gatilao
Jasmine Murcia

TYPESETTER
Star Type
startype@comcast.net

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

www.usenix.org

;login: is the official magazine of the USENIX Association. *;login:* (ISSN 1044-6397) is published 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.

EDITORIAL

Musings

RIK FARROW

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

I just got back from attending the USENIX Annual Technical Conference in Denver. For those of you without gray hairs, USENIX ATC was, for many years, a twice yearly event that drew thousands of attendees for the summer and winter conferences. These were the only USENIX conferences, and the papers covered topics from systems to system administration.

Starting with LISA, new conferences were spun off USENIX ATC to cover specific topic areas: security, file systems, storage, networked systems, and even OSDI, a systems conference. Today, USENIX ATC represents just a shadow of its past, and attendance has dropped from over 3000 to under 300. USENIX ATC is still an important conference for systems researchers.

That's not what I want to write about. USENIX conferences were not considered good material for obtaining tenure, the process whereby an Assistant Professor gains the approval of his or her peers and obtains a lifetime appointment as a full Professor. Margo Seltzer, a past USENIX Board president, worked hard at changing this perception of USENIX conferences, and today USENIX conferences have been given equal weight for evaluation with conferences sponsored by the more traditional CS organizations like ACM and IEEE.

Margo's success did not come without side effects. USENIX conferences were once known for the pragmatic nature of the research accepted and published in their proceedings. As the program committees shifted to a more academic focus, a lot of this pragmatism faded away, replaced by a new pragmatism, one focused instead on publishing papers.

As an outside observer—that is, one not interested in tenure—I was more aware of the change in the tenor of accepted papers. I witnessed a progression to more theoretical work and research that, while interesting, would never be implemented, as the improvements in performance were small, or were only useful in special cases, not for general use.

I also heard grumbling within the ranks. Before USENIX ATC even began this year, I heard people complaining about the quality of some paper reviews. One student suggested that paper submitters be allowed to rank reviewers, just as their own papers were being ranked.

Hakim Weatherspoon mentioned that even though the review process involves blinding, the obscuring of the identities of paper authors, when a group of program committee members tromps out of the PC meeting to avoid a conflict of interest, the remaining PC has very strong hints about the authors of that paper.

Turns out, I hadn't heard anything yet.

The Emperor's Clothes

Bryan Cantrill, the inventor of DTrace and current CTO of Joyent, presented a keynote at USENIX ATC '16 on Thursday morning. He titled his talk, "A Wardrobe for the Emperor: Stitching Practical Bias into Systems Software Research," something that told me little about what was about to unfold. I suggest that you listen to his talk [1], but be aware that it contains some strong language. Much of it is provacative and felt to me like a roast that included a significant number of the senior audience.

Bryan's main point is that program committees for systems conferences have not just veered away from the pragmatic, they've become way too focused on tenure-securing behaviors.

Bryan wasn't taking advantage of a speaking slot to complain about having his papers rejected. He was dramatic for a reason: to stimulate discussion about changing the way the computer science community accepts papers and the purposes for holding conferences. He scheduled a Birds-of-a-Feather (BoF) slot for later the same evening, and showed up to lead the discussion. There were about 30 people in attendance at the start of the BoF, and an hour later people were still talking.

At the BoF, Jon Howell (Google) pointed out that program committees stamp a "quality indicator" on the accepted papers. Another person mentioned the fear of accepting a paper that would be trashed when it appeared at the conference, leading to PCs being less willing to take chances on novel or not-fully-baked research. Bryan mentioned the low acceptance rates, which were once around 25%, but now are less than 20% and sometimes under 15%. Those rates are very low compared to other academic areas, such as microbiology.

Bryan suggested an arXiv (https://arxiv.org/) model, where all papers get "published" and people vote up research that they find the most useful or interesting. Mothy Roscoe thought that this would not work due to the volume of paper submissions, as it was difficult enough to review the smaller number of papers that were submitted to a conference like OSDI (that he co-chairs with Kim Keeton in 2016). Mothy went on to describe how they had structured the OSDI '16 PC, with a two-tiered reviewing system designed to be fair to paper submitters and to reduce the paper-reviewing load on reviewers.

I suggested accepting more papers than there are speaking slots, but only permitting those who could prove their ability to present through the use of a short video. Hakim Weatherspoon, USENIX ATC '16 co-chair, USENIX Board member, and the person who invited Bryan to deliver a keynote address, thought this was impractical. I, and others, pointed out that *every* person presenting at USENIX Security had to create a short video that appeared in the morning Lightning Talks, and using something like this would be a great way to choose those people best at making presentations.

I must confess that the ability to present is a personal sore point for me. I attend paper presentations so I can learn more, and I have learned a lot from the better presenters. The official line on presentations, something I had guessed, and then had confirmed during this BoF, is presentations are not for the purpose of educating your audience. While the presenter certainly does share some information about the paper, the official purpose was to allow audience members to challenge the accepted paper in public during the presentation, putting both the presenter's work, and the decision of the PC, on the line. In reality, this rarely happens, and usually consists of someone complaining that their own paper, with some related work, wasn't cited.

Bryan had also shown a graph of how industry participation in OSDI PCs had dropped from a high of 100% in the late '90s to around 20% currently, resulting in an academically weighted committee. Bryan's definition of a PC member with an "industry connection" was, he stated, very generous, and it would have to be to have reached 100% at any point.

A speech and a BoF about systems conferences and program committees cannot effect immediate change on its own. What Bryan has managed to do, I hope, is to draw attention to the current state of the process by which the systems program committees review and accept papers, and how conferences are organized. Whether we will ever have papers without presentations, accept more papers than there are presentation slots in the program, or use an arXiv-like system instead of the current system are things that steering committees and future program committees will have to decide.

The Lineup

After having written all of that, you might wonder if I've chosen any papers from USENIX ATC '16 for conversion into articles for this issue of *;login:*. Yes, I had picked out a couple of papers, but I also mined Eurosys 2016, as those papers had appeared earlier in the year, and I found some real gems there.

I asked the authors of two related papers if they could write articles about their research. You'll notice that both have quite a practical bent to them. The first paper, by Atlidakis et al., examines how the POSIX standard actually gets used in applications for Debian/Ubuntu, Android, and Mac OS X. What they reveal through their analysis of applications demonstrates the weaknesses of a standard that was created to aid in porting applications between systems over 25 years ago.

Tsai et al. take a different approach, studying how applications use the Linux API. Their work was motivated, in part, by a quest to learn just how much of the API was required to run the most commonly installed Linux system applications, and just how well Linux emulations succeeded at providing a complete Linux API. Some of the same issues, such as use of ioctl(), appear here that appeared in Atlidakis' work.

If you've ever wondered about just how hard it might be to write a useful operating system, you might have come to a thought I'd had decades ago: that the success of an operating system can be measured in its support for the applications that people actually want to use. The operating system itself is secondary as long as the OS offers good performance and reliability. Together these articles, and the papers they are based upon, take a current look at just how hard it is to upset the systems that are currently enthroned. The Tsai paper also points out that people program using frameworks these days, yet operating systems expose very different interfaces.

I invited Jian Xu and Steven Swanson, the authors of a FAST '16 paper on a file system designed for non-volatile memory, to write about their NOVA file system research. Getting another file system into Linux is almost impossible. Yet, non-volatile memories, such as spin memory or Intel/Micron XPoint 3D, require new approaches so that these new persistent storage systems can obtain the maximum potential performance. These memories are not like disks or Flash devices.

Continuing along the theme of systems, I interview Mothy Roscoe again. I had interviewed Mothy six years ago, but wanted to ask about the Barrelfish project, a multikernel OS research project that is still going almost a decade after it first began.

I asked Diego Ongaro, one of the authors of the Raft Consensus Algorithm, presented at USENIX ATC '14, to write more about Raft (https://raft.github.io/), an alternative to Paxos. Diego told me that he wanted to write about Runway, a tool for model checking, simulation, and visualizing the workings of distributed systems. Runway is a work-in-progress, and the Web page (https://runway.systems/) includes a visualization of the Raft Consensus Algorithm in action.

Kalia et al., winner of a Best Paper award at USENIX ATC '16, write about getting the best performance when using RDMA (remote direct memory access). RDMA has long been used in HPC, and is starting to gain some traction in distributed applications. The authors do a good job of explaining how to use RDMA operations for the best performance, something that turns out to be complex but when done well can result in a hundred times faster performance.

I asked Betsy Beyer, an editor of *Site Reliability Engineering* [2], if she would contact the co-authors of one of the *SRE* chapters that really called out to me, the one about the meaning of toil, and reprise it for *;login:*. They did that and more, coming up with a great case study about eliminating toil.

Jonathon Anderson wrote a second part to his article in the summer 2016 issue about routing on Linux systems. The Linux kernel supports multiple default routes, and for multi-homed servers, that's exactly what you want to use.

The Bitcoin blockchain appeals not only to those interested in a non-fiat currency, but also to anyone who would like to take advantage of a system that publishes secure summaries of transactions. Ali et al. present their open source system, Blockstack (blockstack.org), which provides a programming framework for people interested in building systems for name registrations (à la DNS), as well as other uses that rely on a blockchain.

I asked Bruce Potter, one of the founders of SchmooCon (http://shmoocon.org/) and a long-time security expert, to write about how to create a threat model for your own organization. Bruce tells us why a threat model is useful, as well as how you go about creating one and keeping it current.

Dave Beazley reveals one reason why he has been so focused on the Python 3.5 `async` feature. Dave has been writing a module called Curio, and in his column this month he argues for the separation of protocols, such at HTTP/2, and the underlying transport (such as sockets). Curio itself is a library for concurrent I/O (https://curio.readthedocs.io/en/latest/).

Dave Josephsen writes about the emotional stress caused by being on-call. As someone who has spent way too much of his life doing this, Dave was motivated by talks at Monitorama (http://monitorama.com) and wrote a Go program for extracting data using the PagerDuty API. Dave used Go and describes how his program can be easily changed to use other input sources or sinks.

David N. Blank-Edelman exposes us to Consul, an open source distributed key-value system from HashiCorp. The value of Consul goes well beyond Perl users for anyone building distributed services that need a way of finding and updating configuration information on the fly.

Kelsey Hightower makes the case for cleanly shutting down services. Kelsey uses a simple Go program and the `manners` module, which makes it easy to keep a Web service alive until it has processed current requests.

Dan Geer argues for a science of security. He uses the work of T. S. Kuhn as the basis for his argument that paradigms are collections of knowledge representing the current understanding of a branch of science. I liked Dan's discussion, although I wonder if we have even reached the level of engineering when it comes to security. If we built bridges like we build software, no one would dare drive across them.

Robert G. Ferrell maintains his position as commenter-in-chief, and discusses how to learn from distributed computing to make the electric grid truly distributed.

Mark Lamourine has written two reviews about books on Go, and one about Docker, including useful information about the new Docker tools for container orchestration.

Carolyn Rowland, the newly minted USENIX Board President, writes about her experience volunteering for USENIX in USENIX Notes.

Pragmatic Systems Research

USENIX has been known for the high level of pragmatism found in the research published in its conferences. If you read any of the series of history articles found in *;login:* issues from 2015, you would see early *;login*: issues with patches for device drivers included. The UNIX operating system once ran on systems with 32 kilobytes of RAM, and managed to be so useful as to spawn an entire industry.

Today, we have operating systems like Linux, with its 312 system calls, over 700 ioctls and prctls, as well as pseudo-file systems like `/proc` and `/dev` that expose kernel information as files. Yet programmers write using studio tools, which work on top of frameworks, suggesting that only the authors of frameworks need to understand an operating system's interfaces.

I strongly believe that Bryan Cantrill raised some important points in his USENIX ATC '16 keynote. I had heard rumblings about the failures of the current systems many times before. Program Committees are, after all, human institutions, and that means that bias is always an issue, not just a possibility.

Should systems research be published for the purpose of extending tenure to deserving Assistant Professors, or should that be a by-product of creating research that makes the systems we design more useful and secure? Ideally, we will be doing both.

References

[1] Bryan Cantrill's USENIX ATC '16 Keynote: "A Wardrobe for the Emperor: Stitching Practical Bias into Systems Software Research": https://www.usenix.org/conference/atc16/technical-sessions/presentation/cantrill.

[2] B. Beyer, C. Jones, J. Petoff, and N. R. Murphy, eds., *Site Reliability Engineering* (O'Reilly Media, 2016).

SYSTEMS

POSIX Has Become Outdated

VAGGELIS ATLIDAKIS, JEREMY ANDRUS, ROXANA GEAMBASU, DIMITRIS MITROPOULOS, AND JASON NIEH

Vaggelis Atlidakis is a PhD student in the Department of Computer Science at Columbia University. His research interests revolve around broad areas of computer systems, including operating systems, security and privacy, and applications of machine learning to systems. vatlidak@cs.columbia.edu

Jeremy Andrus holds a PhD in operating systems and mobile computing from Columbia University and currently works as a Kernel Engineering Manager at Apple. jeremy_andrus@apple.com

Roxana Geambasu is an Assistant Professor of Computer Science at Columbia University. Her research interests are wide-ranging and include distributed systems, security and privacy, operating systems, databases, and the application of cryptography and machine learning to systems. roxana@cs.columbia.edu

The POSIX standard for APIs was developed over 25 years ago. We explored how applications in Android, OS X, and Ubuntu Linux use these interfaces today and found that weaknesses or deficiencies in POSIX have led to divergence in how modern applications use the POSIX APIs. In this article, we present our analysis of over a million applications and show how developers have created workarounds to shortcut POSIX and implement functionality missing from POSIX.

The Portable Operating System Interface (POSIX) is the IEEE standard operating system (OS) service interface for UNIX-based systems. It describes a set of fundamental abstractions needed for efficient construction of applications. Since its creation over 25 years ago, POSIX has evolved to some extent (e.g., the most recent update was published in 2013 [9]), but the changes have been small overall. Meanwhile, applications and the computing platforms they run on have changed dramatically: modern applications for today's smartphones, desktop PCs, and tablets interact with multiple layers of software frameworks and libraries implemented atop the OS.

Although POSIX continues to serve as the single standardized interface between these software frameworks and the OS, little has been done to measure whether POSIX abstractions are effective in supporting modern application workloads, or whether new, non-standard abstractions are taking form, dethroning traditional ones.

We present the first study of POSIX usage in modern OSes, focusing on three of today's most widely used mobile and desktop OSes—Android, OS X, and Ubuntu—and popular consumer applications characteristic to these OSes. We built a utility called *libtrack*, which supports both dynamic and static analyses of POSIX use from applications. We used *libtrack* to shed light on a number of important questions regarding the use of POSIX abstractions in modern OSes, including which abstractions work well, which appear to be used in ways for which they were never intended, which are being replaced by new and non-standard abstractions, and whether the standard is missing any fundamental abstractions needed by modern workloads. Our findings can be summarized as follows:

First, *usage is driven by high-level frameworks, which impacts POSIX's portability goals*. The original goal of the POSIX standard was application source code portability. However, modern applications are no longer being written to standardized POSIX interfaces. Instead, they rely on platform-specific frameworks and libraries that leverage high-level abstractions for inter-process communication (IPC), thread pool management, relational databases, and graphics support.

Modern OSes gravitate towards a more layered programming model with "taller" interfaces: applications directly link to high-level frameworks, which invoke other frameworks and libraries that may eventually utilize POSIX. This new, layered programming model imposes challenges to application portability and has given rise to many different cross-platform SDKs that attempt to fill the gap left by a standard that has not evolved with the rest of the ecosystem.

Dimitris Mitropoulos is a Postdoctoral Researcher in the Computer Science Department at Columbia University in the City of New York. He holds a PhD ('14) in Cybersecurity with distinction from the Athens University of Economics and Business. His research interests include application security, systems security, and software engineering. dimitris.i.mitropoulos@gmail.com

Jason Nieh is a Professor of Computer Science and Co-Director of the Software Systems Laboratory at Columbia University. Professor Nieh has made research contributions in software systems across a broad range of areas, including operating systems, virtualization, thin-client computing, cloud computing, mobile computing, multimedia, Web technologies, and performance evaluation. nieh@cs.columbia.edu

Second, *extension APIs, namely* `ioctl`, *dominate modern POSIX usage patterns as OS developers increasingly use them to build support for abstractions missing from the POSIX standard.* Extension APIs have become the standard way for developers to circumvent POSIX limitations and facilitate hardware-supported functionality for graphics, sound, and IPC.

Third, *new abstractions are arising, driven by the same POSIX limitations across the three OSes, but the new abstractions are not converging.* To deal with abstractions missing from the aging POSIX standard, modern OSes are implementing new abstractions to support higher-level application functionality. Although these interfaces and abstractions are driven by similar POSIX limitations and are conceptually similar across OSes, they are not converging on any new standard, increasing the fragmentation of POSIX across UNIX-based OSes.

We believe our findings have broad implications related to the future of POSIX-compliant OS portability, which the systems research community and standards bodies will likely need to address in the future. To support further studies across a richer set of UNIX-based OSes and workloads, we make *libtrack*'s source code, along with the application workloads and traces, publicly available at: https://columbia.github.io/libtrack/. The full version of our work was published in EuroSys 2016 [3].

Methodology

Our study involves two types of experiments with real, client-side applications on the three OSes: *dynamic experiments* and *static analysis*. In support of our study, we developed *libtrack*, a tool that traces the use of a given native C library from modern applications. *libtrack* implements two modules: a dynamic module and a static module.

libtrack

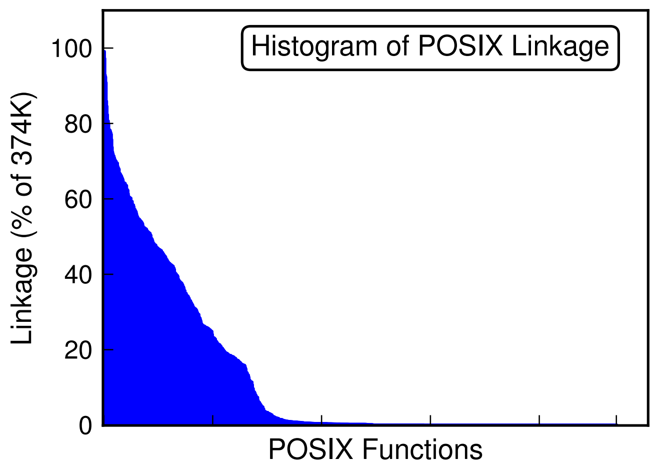
Dynamic Module: *libtrack*'s dynamic module traces all invocations of native POSIX functions for every thread in the system. For each POSIX function implemented in the C standard library of the OS, *libtrack* interposes a special "wrapper" function with the same name, and once a native POSIX function is called, *libtrack* logs the time of the invocation and a backtrace identifying the path by which the application invoked the POSIX function. It also measures the time spent executing the POSIX function. *libtrack* then analyzes these traces to construct a call graph and derive statistics and measurements of POSIX API usage.

Static Module: *libtrack* also contains a static module, which is a simple utility to help identify application linkage to POSIX functions of C standard libraries. Given a repository of Android APKs or of Ubuntu packages, *libtrack*'s static module first searches each APK or package for native libraries. Then it decompiles native libraries and scans the dynamic symbol tables for relocations to POSIX symbols. Dynamic links to POSIX APIs are indexed per application (or per package) and are finally merged to produce aggregate statistics of POSIX linkage.

Workloads

Using *libtrack*, we performed both dynamic and static experiments. We used different workloads for each experiment type centered around consumer-oriented applications; these do not reflect POSIX's standing in other types of workloads, such as server-side or high-performance computing workloads.

Dynamic Experiments: We performed dynamic experiments by interacting with popular Android, OS X, and Ubuntu applications (apps). We selected these apps from the official marketplaces for each OS: Google Play (45 apps), Apple AppStore (10 apps), and Ubuntu Software Center (45 apps). We chose popular apps based on the number of installs, selecting apps across nine categories, and interacted manually with these applications by performing typical operations, such as refreshing an inbox or sending an email with an email application, on commodity devices (laptops and tablets).



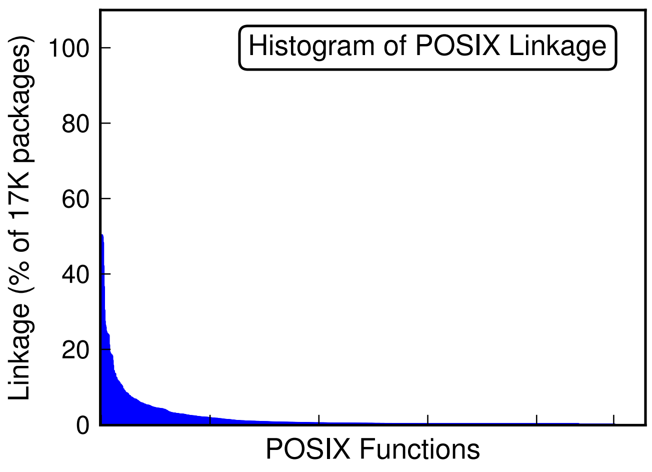


Figure 1: POSIX abstractions linkage

Static Experiments: We performed static experiments of POSIX usage at large scale by downloading over a million consumer applications and checking these applications, and associated libraries, for linkage to POSIX functions of C standard libraries. For Android, we downloaded 1.1 million free Android apps from a Dec. 4, 2014 snapshot of Google Play [10]. For Ubuntu, we downloaded 71,199 packages available for Ubuntu 12.04 on Dec. 4, 2014, using the Aptitude package manager.

Results

We organized the results from our study in a sequence of questions regarding the use of POSIX in modern OS. We began by asking which POSIX functions and abstraction families were being used and which were not being used by modern workloads.

Which Abstractions Are Used and Which Are Not Used by Modern Workloads?

To answer this question, we first used our static analysis on Android and Ubuntu in order to examine which of the implemented abstractions were actually linked by applications or their libraries, giving us a more accurate view into what abstractions are not used by modern workloads. Afterwards, we used results from our dynamic experiments to examine what abstractions are effectively invoked by modern workloads, telling us what the popular POSIX abstractions are.

Linked Abstractions: Figure 1 shows the number of Android apps and Ubuntu packages that dynamically link to POSIX functions of the respective C standard libraries. The results come from our large-scale static analyses of 1.1 million Android apps and 71,989 Ubuntu packages.

Overall, in Android, of the 821 POSIX functions implemented, 114 of them are never dynamically linked from any native library, and approximately half (364 functions) are dynamically linked from 1% or fewer of the apps. Furthermore, our static analysis of Ubuntu packages shows that desktop Linux has a similar, albeit less definitive, trend with Android: phasing out traditional IPC and FS POSIX abstractions.

Dynamically Invoked Abstractions: Although linkage is a definite way to identify unused functions, it is only a speculative way to infer usage. Therefore, we next examine the actual usage of POSIX functions during our dynamic experiments with 45 Android apps. POSIX functions are categorized in abstraction subsystems; the most popular in term of invocations, as well as the most expensive in terms of CPU time, are shown in Figure 2.

We observe that memory is the most heavily invoked subsystem. Typical representatives of this subsystem include user-space utilities for memory handling and system call-backed functions for (de)allocation of memory regions and protection properties setting. As shown in Figure 2, the memory subsystem is also the most expensive subsystem in terms of CPU consumption. The popularity and the cost of memory calls are driven by the proliferation of high-level frameworks that are heavily dependent on memory-related OS functionality. The second most heavily invoked POSIX subsystem is threading. Its popularity is mainly due to Thread Local Storage (TLS) operations, which are very usual in Android to help map between high-level framework threads and low-level native `pthreads`. These operations are relatively lightweight and therefore, contrary to memory, threading accounts for relatively little CPU time. Most of the remaining subsystems include popular or expensive functions, but their CPU time cost is usually proportional to the volume of invocations.

Crucially, there is one surprising exception: `ioctl`. This function alone accounts for more than 16% of the total CPU time, despite its relatively low volume of invocations (0.6%). The striking popularity of `ioctl`—an extension API that lets applications bypass well-defined POSIX abstractions and directly interact with the

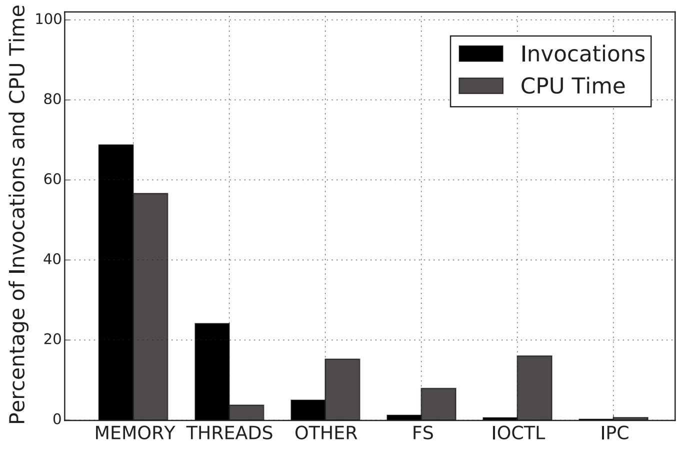


Figure 2: POSIX invocations and CPU consumption for 45 Android apps

kernel—proclaims that POSIX abstractions experience significant limitations in supporting modern application workloads. Therefore, developers increasingly resort to `ioctl` and implement custom support missing from POSIX.

To gain a view on the type of functionality implemented using `ioctl` across the three OSes, we inspect stack traces leading to `ioctl` invocations and identify which libraries triggered them. Table 1 shows the top libraries across the three OSes. In Android, graphics libraries lead to the lion's share of `ioctl` invocations, followed by *Binder*. In OS X, the majority of `ioctl` invocations come from network libraries. In Ubuntu, graphics libraries trigger approximately half of `ioctl` invocations, and the remaining part is mainly due to libraries providing network functionality. We next ask *what* are the abstractions missing from POSIX, if any, and *why* do framework libraries resort to unstructured, extension APIs to implement their functionality?

Does POSIX Lack Abstractions Needed by Modern Workloads?

Taking hints from Table 1, we investigate graphics functionality in modern OSes, which significantly rely on `ioctl` signaling that such abstractions are missing in POSIX. In the following section, we additionally discuss new IPC abstractions, which are replacing older abstractions and are implemented atop `ioctl`.

Graphics: POSIX explicitly omits graphics as a point of standardization. As a result, there is no standard OS interface to graphics cards, and different platforms have chosen different ways to optimize this critical resource. The explicit omission of a graphics standard could be due to the emergence of OpenGL, a popular cross-platform graphics API used in drawing and rendering. While OpenGL works well for applications, OS and graphics library designers have no well-defined interfaces to accomplish increasingly complicated operations.

The lack of a standard kernel interface to GPUs has led to limited extensibility. To alleviate this fragmentation, modern GPU drivers and libraries are trending towards a general purpose computational resource, and POSIX is not in a position to standardize the use of this new, powerful paradigm. These problems have been studied in detail the past four years. For example, the PTask API [8] defines a dataflow-based programming model that allows the application programmer to utilize GPU resources in a more intuitive way, matching API semantics with OS capabilities. This new OS abstraction results in massive performance gains.

The lack of a standard graphics OS abstraction also causes development and runtime problems. With no alternative, driver developers are forced to build their own structure around the only available system call: `ioctl`. Using opaque input and output tokens, the `ioctl` call can be general purpose, but it was never intended for the complicated, high-bandwidth interface GPUs require. Graphics library writers must either use `ioctl` as a funnel into which all GPU command and control is sent via opaque data blobs, or they must design a vendor-specific demux interface akin to the `syscall` system call.

Interestingly, in OS X, graphics functionality does not account for any significant portion of `ioctl` invocations. The reason is that the driver model in OS X, `IOKit`, is more structured than in Android or Ubuntu, and it uses a well-defined Mach IPC interface that can effectively marshal parameters and graphics data across the user-kernel boundary. This creates a reusable, vendor-agnostic API. However, the current interface is designed around the same black-box hardware that runs on other platforms. Therefore, a standardized OS interface to graphics processors would likely have the same benefits on OS X as it would on Ubuntu or Android.

OS	First library	Second library	Third library	Invocations
Android	Graphics (74%) (e.g., libnvrm)	Binder (24%) (e.g., libbinder)	Other (1%)	1.3M
OS X	Network (99%) (e.g., net.dylib)	Loader (0.6%) (e.g., dyld)	—	682
Ubuntu	Graphics (52%) (e.g., libgtk)	Network (47%) (e.g., libQtNet)	Other (1%)	0.4M

Table 1: Top libraries that invoke `ioctl`

Tx/Rx—Android	Pipes avg (µs)	UNIX avg (µs)	Binder avg (µs)
32 bytes	40	54	115
128 bytes	44	56	114
1 page	62	73	93
10 pages	291	276	93
100 pages	2402	1898	94

Tx/Rx—OS X	Pipes avg (µs)	UNIX avg (µs)	Mach avg (µs)
32 bytes	6	11	19
128 bytes	7	51	19
1 page	8	54	12
10 pages	18	175	15
100 pages	378	1461	12

Tx/Rx—Ubuntu	Pipes avg (µs)	UNIX avg (µs)
32 bytes	18	18
128 bytes	20	10
1 page	21	20
10 pages	58	27
100 pages	923	186

Table 2: Latency comparison of different IPC mechanisms

What POSIX Abstractions Are Being Replaced?

Continuing with hints from Table 1, we discuss abstractions that exist in POSIX but appear to be replaced by new abstractions.

Inter-Process Communication: A central IPC abstraction in POSIX is the message queue API (`mq_*`). On all platforms we studied, applications used some alternate form of IPC. In fact, Android is missing the `mq_*` APIs altogether. IPC on all of these platforms has divergently evolved beyond POSIX.

- **IPC in Android:** Binder is the standard method of IPC in Android. It overcomes POSIX IPC limitations and serves as the backbone of Android inter- and intra-process communication. Using a custom kernel module, Binder IPC supports file descriptors passing between processes, implements object reference counting, and uses a scalable multithreaded model that allows a process to consume many simultaneous requests. In addition, Binder leverages its access to processes' address space and provides fast, single-copy transactions. Binder exposes IPC abstractions to higher layers of software, and Android apps can focus on logical program flow and interact with Binder through standard Java objects and methods, without the need to manage low-level IPC details. Because no existing API supported all the necessary functionality, Binder was implemented using `ioctl` as the singular kernel interface. Binder IPC is used in every Android application and accounts for nearly 25% of the total measured POSIX CPU time that funnels through the `ioctl` call.
- **IPC in Linux:** The D-Bus protocol [5] provides apps with high-level IPC abstractions in GNU/Linux. It describes an IPC messaging bus system that implements (1) a system daemon monitoring system-wide events and (2) a per-user login session daemon for communication between applications within the same session. The applications we inspect use mostly the *libdbus* implementation of D-Bus (38 out of 45 apps). An evolution of D-Bus, called the Linux Kernel D-Bus, or *kdbus*, is also gaining increasing popularity in GNU/Linux OSes. It is an in-kernel implementation of D-Bus that uses Linux kernel features to overcome inherent limitations of user-space D-Bus implementations. Specifically, it supports zero-copy message passing between processes, and it is available at boot allowing Linux security to directly leverage it.
- **IPC in OS X:** IPC in OS X diverged from POSIX since its inception. Apple's XNU kernel uses an optimized descendant of CMU's Mach IPC [2, 7] as the backbone for inter-process communication. Mach comprises a flexible API that supports high-level concepts such as: object-based APIs abstracting communication channels as *ports*, real-time communication, shared memory regions, RPC, synchronization, and secure resource management. Although flexible and extensible, the complexity of Mach has led Apple to develop a simpler higher-level API called *XPC*. Most apps use XPC APIs that integrate with other high-level APIs, such as Grand Central Dispatch.

To highlight key differences in POSIX-style IPC and newer IPC mechanisms, we adapt a simple Android Binder benchmark from the Android code base to measure both pipes and UNIX domain sockets as well as Binder transactions. We also use the MPMMTest application from Apple's open source XNU [1]. We measure the latency of a round-trip message using several different message sizes, ranging from 32 bytes to 100 (4096 byte) pages. We run our benchmarks using an ASUS Nexus-7 tablet with Android 4.3 Jelly Bean, a MacBook Air laptop (4-core Intel CPU @2.0 GHz, 4 GB RAM) with OS X Yosemite, and a Dell XPS laptop (4-core Intel CPU @1.7 GHz, 8 GB RAM) with Ubuntu 12.04 Precise Pangolin.

The results, averaged over 1000 iterations, are summarized in Table 2. Both Binder and Mach IPC leverage fast, single-copy and zero-copy mechanisms, respectively. Large messages in both Binder and Mach IPC are sent in near-constant time. In contrast, traditional POSIX mechanisms on all platforms suffer from large variation and scale linearly with message size.

Asynchronous I/O: Our experiments with Android, OS X, and Ubuntu apps reveal another evolutionary trend: the replacement of POSIX APIs for asynchronous I/O with new abstractions built atop POSIX multithreading abstractions. The nature and purpose of threads has been a debate in OS research for a long time [4, 6], and POSIX makes no attempt to prioritize a threading model over an event-based model.

Our study reveals that while POSIX locking primitives are still extremely popular, new trends in application abstractions are blurring the line between event and thread by combining high-level language semantics with pools of threads fed by event-based loops. This new programming paradigm is enforced by the nature of GUI applications that require low-latency. While an event-based model may seem the obvious solution, event processing in the input or GUI context leads to suboptimal user experience. Therefore, dispatching event-driven work to a queue backed by a pool of pre-initialized threads has become a de facto programming model in Android, OS X, and Ubuntu. Although this paradigm appears in all the OSes we studied, the implementations are extremely divergent.

Android defines several Java classes that abstract the concepts of creating, destroying, and managing threads (`ThreadPool`), looping on events (`Looper`), and asynchronous work dispatching (`ThreadPoolExecutor`). Ubuntu applications can choose from a variety of libraries providing similar functionality, but through vastly different interfaces. For example, the `libglib`, `libQtCore`, and `libnspr` all provide high-level thread abstractions based on the GNOME desktop environment. In OS X the APIs are, yet again, different. The vast majority of event-driven programming in OS X is done through Mach IPC. Apple has written high-level APIs around event handling, thread pool management, and asynchronous task dispatch. Most of these APIs are enabled through Grand Central Dispatch (GCD). GCD manages a pool of threads and even defines POSIX alternatives to semaphores, memory barriers, and asynchronous I/O. The GCD functionality is exported to applications from classes such as `NSOperation`.

In summary, driven by the strong need for asynchrony and the event-based nature of modern GUI applications, different OSes have created similar but not converging and non-standard adherent-threading support mechanisms.

Conclusion

Perfect application portability across UNIX-based OSes is clearly beyond the realm of possibility. However, maintaining a subset of harmonized abstractions is still a viable alternative for preserving some uniformity within the UNIX-based OSes. Our study shows that new abstractions, beyond POSIX, are taking form in three modern UNIX-based OSes (Android, OS X, and Ubuntu), and that changes are not converging to any new unified set of abstractions. We believe that a new revision of the POSIX standard is due, and we urge the research community to investigate what that standard should be. Our study provides a few examples of abstractions—such as graphics, IPC, and threading—as starting points for re-standardization, and we recommend that changes should be informed by popular frameworks that have a principal role in modern OSes.

Availability

The extended version of our work was published in EuroSys 2016 [3]. To support further studies across a richer set of UNIX-based OSes and workloads, we open source our code along with the application workloads and traces available: https://columbia.github.io/libtrack/.

Acknowledgments

This work was supported in part by DARPA Contract FA8650-11-C-7190, NSF grants CNS-1351089, CNS-1162447, CNS-1422909, and CCF-1162021, a Google Research Award, and a Microsoft Research Faculty Fellowship.

References

[1] Apple, Inc., OS X 10.11.2, Source: http://opensource.apple.com/tarballs/xnu/xnu-3248.20.55.tar.gz, accessed 03/20/2016.

[2] Apple, Inc., Mach Overview: https://developer.apple.com/library/mac/documentation/Darwin/Conceptual/KernelProgramming/Mach/Mach.html, Aug. 2013, accessed 03/22/2015.

[3] V. Atlidakis, J. Andrus, R. Geambasu, D. Mitropoulos, and J. Nieh, "A Measurement Study of POSIX Abstractions in Modern Operating Systems: The Old, the New, and the Missing," in *Proceedings of the 11th European Conference on Computer Systems (EuroSys 2016)*, April 2016.

[4] R. V. Behren, J. Condit, and E. Brewer, "Why Events Are a Bad Idea (for High-Concurrency Servers)," in *Proceedings of the 9th Conference on Hot Topics in Operating Systems,* vol. 9, May 2003, pp. 19–24: https://www.usenix.org/legacy/events/hotos03/tech/full_papers/vonbehren/vonbehren.pdf.

[5] freedesktop.org., D-Bus: http://www.freedesktop.org/wiki/Software/dbus/, accessed 10/18/2015.

[6] J. K. Ousterhout, "Why Threads Are a Bad Idea (for Most Purposes)," presentation given at the 1996 USENIX Annual Technical Conference, Jan. 1996.

[7] R. Rashid, R. Baron, A. Forin, R. Forin, D. Golub, M. Jones, D. Julin, D. Orr, and R. Sanzi, "Mach: A Foundation for Open Systems," in *Proceedings of the 2nd Workshop on Workstation Operating Systems*, IEEE, Sept. 1989, pp. 109–112.

[8] C. J. Rossbach, J. Currey, M. Silberstein, B. Ray, and E. Witchel, "PTask: Operating System Abstractions to Manage GPUs as Compute Devices," in *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP 2011)*, Oct. 2011, pp. 233–248.

[9] The Open Group and IEEE, last POSIX revision: http://pubs.opengroup.org/onlinepubs/9699919799, accessed 03/19/2015.

[10] N. Viennot, E. Garcia, and J. Nieh, "A Measurement Study of Google Play," in *Proceedings of the ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS 2014)*, June 2014, pp. 221–233.

What to Support When You're Supporting
A Study of Linux API Usage and Compatibility

CHIA-CHE TSAI, BHUSHAN JAIN, NAFEES AHMED ABDUL, AND DONALD E. PORTER

Chia-Che Tsai is a PhD student at Stony Brook University. His research involves restructuring operating system designs for less vulnerability and higher performance. He is also the main contributor to the Graphene library OS. chitsai@cs.stonybrook.edu

Bhushan Jain is a PhD student at the University of North Carolina at Chapel Hill. His research interests include virtualization security, memory isolation, and system security. bhushan@cs.unc.edu

Nafees Ahmed Abdul is a Software Engineer at Symbolic IO. He earned his master's degree from Stony Brook University. He is broadly interested in file systems and storage technologies. nabdul@cs.stonybrook.edu

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

Application programming interfaces (APIs) specify how application developers interact with systems. As APIs evolve over the life of a system, the system developers have little in the way of empirical techniques to guide decisions such as deprecating an API. We propose metrics for evaluating the importance of system APIs, as well as the relative maturity of a prototype system that claims partial compatibility with another system. Using these metrics, we study Linux APIs—such as system calls, `ioctl` opcodes, pseudo-files, and libc functions—yielding insights for developers and researchers.

System developers routinely make design choices based on what they believe to be the common and uncommon behaviors of a system. Imagine a developer who is building a prototype system designed to run Linux applications. This developer will prioritize the implementation tasks based on what he or she believes to be important, which may be heavily skewed toward the developer's preferred workloads.

In general, developers struggle to evaluate the impact of adding or removing APIs on backward-compatibility with existing applications, primarily because of a lack of metrics. The state-of-the-art is *bug-for-bug compatibility*, which requires all behaviors (even undefined or undocumented behaviors) of a system to be identical to its predecessor. For instance, deprecating or retiring an API in Linux requires a lengthy process of repeatedly warning application developers to adopt a replacement API and confirming that no applications are broken by the change—a process that can take many years.

In evaluating compatibility or completeness of a prototype system, a common metric is a simple count of supported system APIs. For instance, the Graphene library OS [1] offers support for 143 out of 318 Linux x86-64 system calls. Although system call counts are easy to measure, this metric fails to capture essential aspects of compatibility, such as the fraction of applications or users that could plausibly use the system. In order to indicate general usefulness, a good compatibility metric should relate to the application usage patterns of end users, factoring in both common and uncommon cases.

At the root of these problems is a lack of data sets and analysis of how system APIs are used in practice. System APIs are simply not equally important: some APIs are used by popular libraries and, thus, by essentially every application. Other APIs may be used only by applications that are rarely installed. Evaluating compatibility is fundamentally a measurement problem.

This article summarizes a study of Linux APIs; a longer version is published in EuroSys 2016 [2]. This study contributes a data set and analysis tool that can answer several practical questions about API usage and compatibility. For instance, if a developer were to add one additional API to a given system prototype, which API would most increase the range of supported applications? Or if a given system API is optimized, what widely used applications would likely benefit? Similarly, this data and toolset can help OS maintainers evaluate the impact of an API change on applications and can help users evaluate whether a prototype system is suitable for their needs.

Some APIs Are More Equal than Others

We started this study from a research perspective, in search of a better way to evaluate the completeness of system prototypes. In general, compatibility treated as a binary property (i.e., bug-for-bug compatibility) loses important information when evaluating a prototype that is almost certainly incomplete. Metrics such as the count of supported system APIs are noisy at best and give no guidance as to which APIs are the most important.

One way to understand the completeness of a system or the importance of an API is to measure the impact on end users. In other words, if a system supports a set of APIs, how many applications chosen by users can run on the system? Or if an API were not supported, how many users would notice its absence? To answer these questions, we must consider both the difference in API usage among applications, and application popularity among users. We measure the former by analyzing application binaries and determine the latter from the installation statistics collected by the Debian and Ubuntu Popularity Contests [3, 4].

We introduce two new metrics: one for each API and one for a whole system. For each API, we measure how disruptive its absence would be to applications and end users—a metric we call *API importance*. For a system, we compute a weighted percentage we call *weighted completeness*. For simplicity, we define a *system* as a set of implemented or emulated APIs, and assume an application will work on a target system if the APIs used by the application (or API footprint) is implemented on the system (i.e., we assume implemented APIs work as expected). The popularity of applications is measured by *installations*, which are collections of applications installed by users on physical machines, virtual machines, containers, or partitions in multi-boot systems.

API Importance
For a given API, the probability that an installation includes at least one application requiring the API

API importance indicates how indispensable a given API is to at least one application on a randomly selected installation. Intuitively, if an API is used by no packages or installations, the API importance will be zero and its absence will cause no negative effects. If an API is only used by packages A, B, and C, the API importance will be the probability that either A, B, or C is chosen in an installation, which can be determined from package installation statistics. We consider an API to be *important* to an installation as long as one installed package requires the API. For instance, the `reboot` system call has almost 100% importance, but on most systems, this API is used only by the `/sbin/reboot` binary.

Weighted Completeness
For a target system, the fraction of applications supported, weighted by the popularity of these applications

Weighted completeness indicates the fraction of installed applications that a prototype system can support on a randomly selected installation. Intuitively, a system with bug-for-bug compatibility will be able to support *all* applications and have 100% weighted completeness. If a system only supports packages A, B, and C, the weighted completeness will be the weighted fraction of A, B, and C over the average number of installed packages.

Data Collection

This study focuses on Ubuntu/Debian Linux as a baseline for comparison. We use static analysis to identify the API footprint of all applications and use installation statistics to evaluate the popularity of each application.

The methodology for measuring API importance and weighted completeness is summarized as follows:

1. For each available package, collect the API footprint of the package by disassembling all the binaries. The API footprint includes the APIs called by an executable or called by a library through function calls from the executable.
2. Calculate the API importance of each API based on the packages that use the API as well as the popularity of these packages.
3. For a target system, identify a list of supported APIs of the target system either from the system's source or as provided by the developers of the system.
4. Based on the API footprint of the packages, list the supported and unsupported packages for the target system.
5. Finally, weigh the list of supported packages based on their popularity and calculate the weighted completeness of the target system.

The Scope of This Study

Types of system APIs: We study various types of APIs defined in x86-64 Linux 3.19:

- 318 defined system call numbers.
- Opcodes for vectored system calls (635 `ioctl` opcodes, 18 `fcntl` opcodes, and 44 `prctl` opcodes).
- Pseudo-files in `proc`, `dev`, and `sys` file systems. In our application sample, we found 5,846 unique, hard-coded paths.
- 1,274 global functions in GNU libc 2.21 (libc.so only).

Sample of applications: 30,976 packages downloaded through `APT` on Ubuntu Linux 15.04.

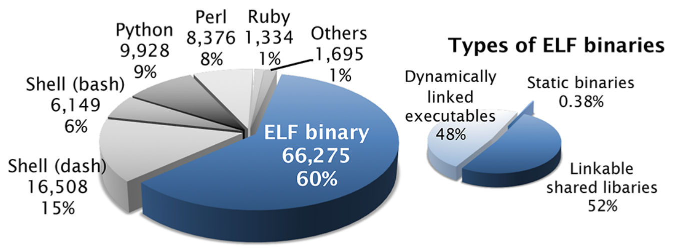


Figure 1: Types of applications and ELF binaries studied

Figure 1 shows the types of applications in these packages. We focused primarily on ELF binaries, which account for the largest fraction (60%) of Linux applications. For interpreted languages such as shell languages (21%) or Python (9%), we assume the API footprint of the applications is covered by the API footprint of the interpreter.

Package installation statistics: 2,935,744 installations collected in the Ubuntu and Debian Popularity Contests.

API Importance of Linux System Calls

We begin by looking at the API importance of each Linux system call, in order to answer the following questions:

- Which system calls are the most important to support in a new system or have highest costs to replace?
- Which system calls are candidates for deprecation?
- Which system calls are not supported by the OS but are still attempted by applications?

There are 318 system call numbers defined in x86-64 Linux 3.19. Figure 2 shows the distribution of system calls by API importance, ordered from the most important (near 100%) to the least important (0%)—similar to an inverted CDF.

Our study shows that over two-thirds (224 of 318) of Linux system calls have nearly 100% API importance. These system calls are indispensable for users—required by at least one application on every installation. Therefore, changing or removing these system calls would be highly disruptive.

On the other hand, we found 44 system calls with API importance above zero but less than 10%. In some cases, there are more popular alternatives with overlapping functionality. For instance, API importance for the System V message queue system calls (e.g., 100% for `msgget`) is higher than for POSIX message queue system calls (e.g., 5% for `mq_open`), although Linux supports both. This is attributable to System V message queues being more portable to other UNIX systems. Sometimes comparable functionality is provided by pseudo-files. For instance, the information returned by the system call `query_module` is also available by reading the pseudo-files `/proc/modules` and `/proc/kallsyms`.

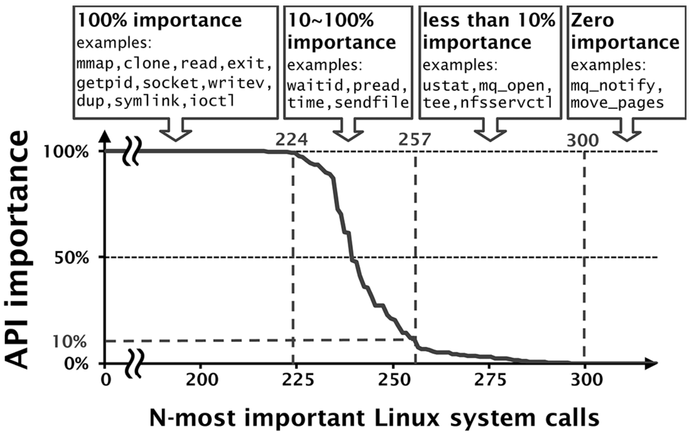


Figure 2: API importance of 318 Linux system calls in x86-64 Linux 3.19, ordered from the most to the least important

We also found five system calls—`uselib`, `nfsservctl`, `afs_syscall`, `vserver`, and `security`—that are officially retired but still have a non-zero API importance. These system calls are used because some applications still attempt the old calls for backward-compatibility with older kernels and, if these calls are unsupported, attempt newer API variants.

In total, 18 of 318 system calls defined in Linux 3.19 are not explicitly used by any application we studied. Eleven of these system calls are defined but retired and thus do not have an entry point in the kernel. Six system calls (`rt_tgsigqueueinfo`, `get_robust_list`, `remap_file_pages`, `mq_notify`, `lookup_dcookie`, and `move_pages`) are available but not used by any applications. These system calls are potential candidates for deprecation.

From "Hello World" to Every Application

For prototype systems or emulation layers, API importance and weighted completeness are useful for determining which APIs to implement first and for evaluating the progress of the prototypes. Our study shows an optimal path for adding system calls to a prototype system using a simple, greedy strategy of implementing the most important APIs first, which in turn maximizes weighted completeness.

Table 1 and Figure 3 demonstrate the optimal path of implementing Linux system calls, split into five stages, and the upper bound of weighted completeness that can be achieved. Essentially, one cannot run even the most simple application in Ubuntu/Debian Linux without at least 40 system calls. After this, the number of additional applications one can support by adding another system call increases steadily up to an inflection point at 125 system calls, or supporting extended attributes on files, where weighted completeness jumps to 25%. To support roughly half of Ubuntu/Debian Linux applications, one must have 145 system calls, and the curve plateaus around 202 system

Stage	System call examples	No. of system calls to support	Weighted completeness
I	`mmap, vfork, exit, read, fcntl, kill, dup2`	40	1.12%
II	`mremap, ioctl, access, socket, poll, recvmsg`	+41 (81)	10.68%
III	`shutdown, symlink, alarm, listen, shmget, pread64`	+64 (145)	50.09%
IV	`flock, semget, ppoll, mount, brk, pause, clock_gettime`	+57 (202)	90.61%
V	All remaining	+70 (272)	100%

Table 1: The optimal path of adding system calls in prototype systems, based on the order of API importance, to optimize the accumulated weighted completeness

calls. We do not provide a complete ordered list here in the interest of brevity, but this list is available as part of our released data set.

One of the uses of weighted completeness is to help guide the process of developing new prototype systems. Because 224 out of 318 system calls on Ubuntu/Debian Linux have 100% API importance, if one of these 224 calls is missing, at least one application on a typical system will not work. Weighted completeness, however, is more forgiving, as it tries to capture the fraction of a typical installation that could work. Only 40 system calls are needed to support at least one application and have weighted completeness of more than 1%.

For simplicity, the optimal path we recommend only includes system calls, but one can construct a similar path including other types of APIs, such as vectored system calls, pseudo-files, and library functions.

Weighted Completeness of Linux Systems

We evaluate the weighted completeness of four systems or emulation layers: User-Mode-Linux [5], L4Linux [6], the FreeBSD's Linux emulation layer [7], and the Graphene library OS [1]. For each system, we identify the supported system calls by examining the defined system call tables in the source code. Figure 4 shows the weighted completeness of these systems based on the supported system calls. User-Mode-Linux (UML) and L4Linux both have over 90% weighted completeness, with more than 280 system calls implemented. FreeBSD's weighted completeness is 62.3% due to missing some less important system calls (e.g., `inotify_init`). Graphene's weighted completeness is only 0.42% due to missing scheduling control, but this is improved to 21.1% by adding two scheduling system calls.

For prototype developers, the proposed optimal path can maximize weighted completeness on a limited development budget,

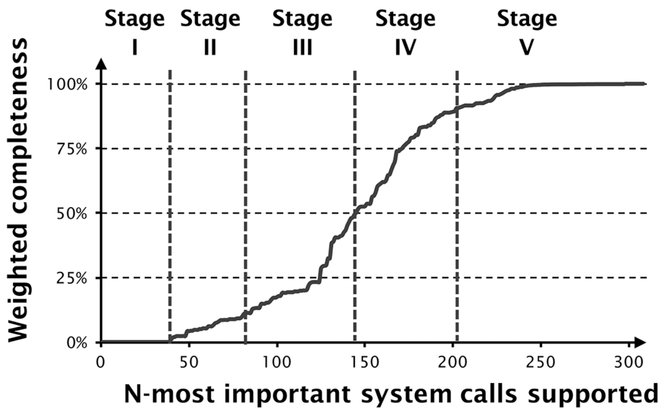


Figure 3: Accumulated weighted completeness when *N*-most important system calls are supported in the prototype system

especially for systems that implement a smaller fraction of APIs, such as Graphene or FreeBSD's Linux emulation layer. For existing prototypes, our study can identify the most important APIs that are missing, but can boost the weighted completeness most, witness the dramatic improvement (0.42% to 21.1%) on Graphene.

Vectored System Call Opcodes

Some system calls, such as `ioctl`, `fcntl`, and `prctl`, essentially export a secondary system call table using the first argument as an operation code (*Opcode*). These *vectored system calls* significantly expand the system API, which we also consider in evaluating compatibility.

Among all the vectored system calls, `ioctl` represents the largest expansion of the Linux system APIs. There are 635 `ioctl` opcodes defined in the Linux 3.19 kernel source alone, and other kernel modules and drivers developed by third parties can define additional opcodes. Figure 5 shows the API importance of `ioctl` system call opcodes defined in Linux 3.19, ordered from the most important to the least important.

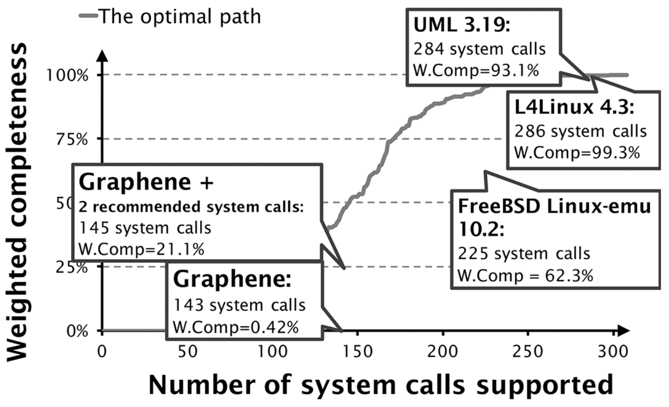


Figure 4: Weighted completeness of four systems or emulation layers with partial compatibility to Linux

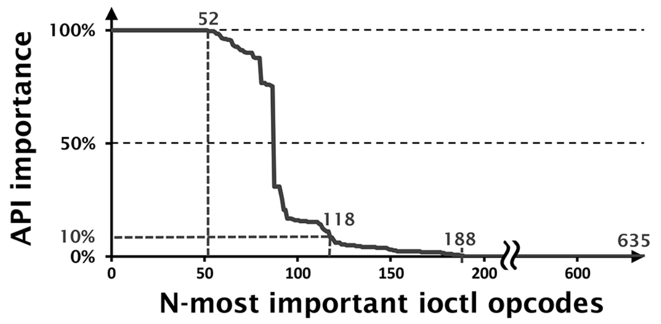


Figure 5: API importance of 635 `ioctl` opcodes defined in Linux 3.19 source, ordered from the most to the least important

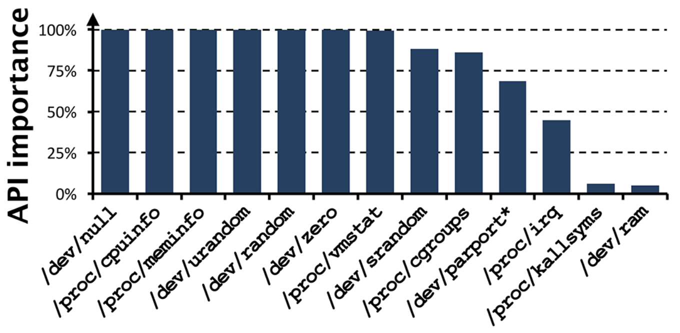


Figure 6: API importance of selected pseudo-files and devices under `/proc` and `/dev`

We observe a different trend for the API importance of `ioctl` opcodes from the system calls. Among the 635 `ioctl` opcodes defined in the Linux kernel source, fewer than one-tenth (52 opcodes) are indispensable on every installation, whereas more than two-thirds (447 opcodes) are never used by any application in Ubuntu/Debian Linux. In other words, when system developers implement `ioctl` opcodes in their prototype systems, it is more productive to focus on the opcodes that have higher API importance and implement the rest as needed for specific applications.

For the opcodes of other two vectored system calls, `fcntl` and `prctl`, we observe that the fraction of their opcodes being indispensable on every installations is higher than the `ioctl` opcodes. In Linux 3.19, 18 `fcntl` opcodes and 44 `prctl` opcodes are defined. Among them, 11 `fcntl` opcodes and nine `prctl` opcodes have 100% API importance.

Pseudo-Files and Devices

In addition to the main system call table, Linux exports many additional APIs through pseudo-file systems, often mounted at `/proc`, `/dev`, and `/sys`. These are called pseudo-files because they are not backed by any physical storage but instead export the contents of kernel data structures to applications or administrators. Although many of these pseudo-files are used on the command line or in scripts input by an administrator, there is also routine use of pseudo-files in applications.

We use static analysis to find hard-coded pseudo-file paths in application binaries. Our approach does not capture the cases where paths of pseudo-files are passed in as input to the applications, such as `dd if=/dev/zero`. However, we observe that when pseudo-files are widely used as alternative system APIs, their paths tend to be hard-coded in the binary as a string or string pattern, such as `/proc/%d/cmdline`, where `%d` can be any process ID. Our analysis captures hard-coded paths and string patterns.

Easy extensibility is an appeal of using pseudo-files as system APIs; as a result, their count can be an order-of-magnitude larger than the number of system calls or opcodes. We found 12,039 binaries that access pseudo-files and devices. Of these, 5846 unique paths were found hard-coded in these binaries. Figure 6 shows the API importance of several common paths for pseudo-files and devices.

We find that several files, such as `/dev/null`, are widely used through hard-coded paths in applications, even though there might be simpler alternatives. For instance, among 12,039 binaries that use a hard-coded path, 3324 are hard-coded to access `/dev/null`. Although `/dev/null` is convenient for use on the command line and in scripts, it is surprising that such a significant number of applications write to this pseudo-file rather than eliding the `write` system call.

Because many pseudo-files are accessed from the command line, it is hard to conclude that any should be deprecated. Nonetheless, these files represent large and complex APIs that create an important attack surface to defend. As noted in other studies, the permissions on pseudo-files and devices tend to be set liberally enough to leak a significant amount of information [8]. For files used by a single application, an abstraction like a fine-grained capability [9] might better capture the security requirement.

Standard System Library Functions

Functions defined in standard system libraries, such as libc, are also system APIs. It is a common practice that developers tend to use library functions as more portable, user-friendly wrappers of the kernel APIs instead of directly calling these kernel APIs. For instance, GNU libc [10] exports functions for using locks and condition variables, which internally use the more subtle `futex` system call.

We study 1274 global function symbols exported by GNU libc 2.21 (`libc.so` only; other libraries, such as `libm.so` and `libpthread.so` are not included). Among these functions, 42.8% have an API importance of 100%, 50.6% have a API importance of less than 50%, and 39.7% have an API importance of less than 1%, including ones that are never used. This result implies that processes load a significant amount of unnecessary code into their address space. By splitting libc into sub-libraries based on API importance and common linking patterns, systems could realize a non-trivial space savings and reduce the attack surface for code reuse attacks.

We analyzed the space savings of a GNU libc 2.21 which removed any APIs with API importance lower than 90%. In total, libc would retain 889 APIs and the size would be reduced to 63% of its original size. The probability that an application would need a missing function and load it from another library is less than 9.3% (equivalent to 90.7% weighted completeness for the stripped libc). Further decomposition is also possible, such as placing APIs that are commonly accessed by the same application into the same sub-library.

Unweighted API Usage in Applications

Weighing metrics based on installations is important for understanding the impact of API changes on end users. By removing this weight, however, we can also observe trends in how APIs are used by application developers.

Unweighted API Importance
For a given API, the probability an application (package) uses that API, regardless of its installation probability

Unweighted API importance shows the preference of application developers for an API over its variants and the effort to communicate with all relevant application developers if an API is changed.

One common reason for system developers to design API variants is to replace APIs that are prone to security problems. For instance, many of the `set*id` system calls (e.g., `setuid`) have subtle semantic differences across UNIX platforms. Chen et al. [11] conclude that `setresuid` is the most secure choice for having the clearest semantics across all UNIX flavors. Another example is that file-accessing system calls (e.g., `access`) can be exploited through time-of-check-to-time-of-use (TOCTTOU) attacks, and their variants (e.g., `faccessat`) can be used to resist these attacks.

Table 2 shows the difference in unweighted API importance among the secure and insecure API variants. In some cases, like `set*id` system calls, the secure API variants (e.g., `setresuid`) are well-adopted and have higher unweighted API importance than the insecure ones. However, in more cases, like `get*id` and `access`, the insecure API variants are more commonly used by application developers.

Besides security-related reasons, system developers may create API variants by retaining old APIs for backward-compatibility. For instance, the `wait4` system call is considered obsolete and will soon be replaced by `waitid` [12], but `wait4` still remains available in Linux. In some other cases, multiple API variants are retained because one variant is specific to a particular OS like Linux, and the other is more generic and portable. Table 3 shows the difference in unweighted API importance among these variants. In general, API variants designed to be portable (e.g., `writev`) are more commonly used by the application developers than Linux-specific ones (e.g., `pwritev`). However, older APIs (e.g., `wait4`) may still be widely used in applications, even though the new APIs are introduced to improve the application portability.

Insecure API	Usage	Secure API	Usage
`setuid`	15.67%	`setresuid`	99.68%
`setreuid`	1.88%		
`getuid`	99.81%	`getresuid`	36.19%
`geteuid`	55.15%		
`access`	74.24%	`faccessat`	0.63%
`rename`	43.18%	`renameat`	0.30%
`chmod`	39.80%	`fchmodat`	0.13%

Table 2: Usage (unweighted API importance) of secure and insecure API variants in applications

Old (Obsolete) APIs vs. New APIs

Old API	Usage	New API	Usage
`getdents`	99.80%	`getdents64`	0.08%
`tkill`	0.51%	`tgkill`	99.80%
`wait4`	60.56%	`waitid`	0.24%

Linux-Specific APIs vs. Portable APIs

Linux-specific API	Usage	Portable API	Usage
`preadv`	0.15%	`readv`	62.23%
`pwritev`	0.16%	`writev`	99.80%
`accept4`	0.93%	`accept`	29.35%
`recvmmsg`	0.11%	`recvmsg`	68.82%
`sendmmsg`	5.17%	`sendmsg`	42.49%

Table 3: Usage (unweighted API importance) of similar API variants in applications

Conclusion

In this study, we define new metrics for evaluating API usage and compatibility, and, based on these results, we draw several conclusions about the nature of Linux APIs. First, for any OS installation in our data set, the required API size is several times larger than the all-system calls defined in Linux, once one considers vectored system call opcodes and pseudo-files. We also show that a substantial range of system calls and other APIs are rarely used. Finally, we provide a method to evaluate partial support of APIs in prototype systems, and plot an optimal path for adding system calls. We expect that the data set will be of use to researchers and developers for further study, and the methodology can be applied to future releases and other operating systems.

The data set and analysis tool are available at: http://oscar.cs.stonybrook.edu/api-compat-study.

Acknowledgments

We thank Bianca Schroeder and William Jannen for their insightful comments on this work. This research was supported in part by NSF grants CNS-1149229, CNS-1161541, CNS-1228839, CNS-1405641, CNS-1408695, CNS-1526707, and VMware. Bhushan Jain is supported by an IBM PhD Fellowship.

References

[1] C. Tsai, K. S. Arora, N. Bandi, B. Jain, W. Jannen, J. John, H. A. Kalodner, V. Kulkarni, D. Oliveira, and D. E. Porter, "Cooperation and Security Isolation of Library OSes for Multi-Process Applications," in *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, 2014.

[2] C. Tsai, B. Jain, N. Ahmed Abdul, and D. E. Porter, "A Study of Modern Linux API Usage and Compatibility: What to Support When You're Supporting," in *Proceedings of the ACM European Conference on Computer Systems (EuroSys),* 2016.

[3] Ubuntu popularity contest: http://popcon.ubuntu.com.

[4] Debian popularity contest: http://popcon.debian.org.

[5] J. Dike, *User Mode Linux* (Prentice Hall, 2006).

[6] H. Härtig, M. Hohmuth, J. Liedtke, J. Wolter, and S. Schonberg, "The Performance of μ-Kernel-Based Systems," *SIGOPS Operating System Review,* vol. 31, no. 5 (Dec. 1997), pp. 66–77.

[7] R. Divacky, "Linux Emulation in FreeBSD," master's thesis: http://www.freebsd.org/doc/en/articles/linux-emulation.

[8] S. Jana and V. Shmatikov, "Memento: Learning Secrets from Process Footprints," in *Proceedings of the IEEE Symposium on Security and Privacy*, 2012.

[9] J. S. Shapiro, J. M. Smith, and D. J. Farber, "EROS: A Fast Capability System," in *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles (SOSP),* 1999.

[10] The GNU C library: http://www.gnu.org/software/libc.

[11] H. Chen, D. Wagner, and D. Dean, "Setuid Demystified," in *Proceedings of the 11th USENIX Security Symposium,* 2002.

[12] `wait4(2)` Linux man page: http://linux.die.net/man/2/wait4.

NOVA
A Log-Structured File System for Hybrid Volatile/Non-Volatile Main Memories

JIAN XU AND STEVEN SWANSON

Jian Xu is a PhD candidate in the Department of Computer Science and Engineering at the University of California, San Diego. His research interests include operating system and system software design for next-generation storage technologies. He is working together with Professor Steven Swanson in the Non-Volatile Systems Laboratory. jix024@cs.ucsd.edu

Steven Swanson is a Full Professor in the Department of Computer Science and Engineering at the University of California, San Diego and the Director of the Non-Volatile Systems Laboratory. His research interests include systems, architecture, security, and reliability issues surrounding non-volatile, solid-state memories. He has also co-led projects to develop low-power co-processors for irregular applications and to devise software techniques for using multiple processors to speed up single-threaded computations. In previous lives he has worked on scalable dataflow architectures, ubiquitous computing, and simultaneous multithreading. He received his PhD from the University of Washington in 2006. swanson@cs.ucsd.edu

NOVA is a new kind of log-structured file system designed for emerging non-volatile main memory (NVMM) technologies, like Intel's 3D XPoint memory. NOVA provides better performance and stronger consistency guarantees than either conventional block-based file systems or other, recently proposed NVMM file systems. NOVA's focus on NVMMs leads to three key design decisions: NOVA maintains a separate metadata log for each file and directory, uses copy-on-write to provide write atomicity, and applies lightweight journaling to make complex operations atomic. These techniques allow NOVA to improve performance by a factor of between 3.1 and 13.5 without jeopardizing crash consistency.

Emerging non-volatile memory (NVM) technologies such as spin-torque transfer, phase change, resistive memories [2, 8], and Intel and Micron's 3D Xpoint [1] technology promise to revolutionize I/O performance. Researchers have proposed placing NVMs on the processor's memory bus alongside conventional DRAM, leading to hybrid volatile/non-volatile main memory systems [12]. Combining faster, volatile DRAM with slightly slower, denser non-volatile main memories (NVMMs) offers the possibility of storage systems that combine the best characteristics of both technologies.

Hybrid DRAM/NVMM storage systems present a host of opportunities and challenges for system designers. These systems should improve conventional file access performance and allow applications to abandon slow read/write file interfaces in favor of faster memory-mapped, load/store access interfaces. They will also allow for increased concurrency and efficiently support more flexible access patterns. File systems must realize these advantages while still providing the strong consistency guarantees that applications require and respecting the limitations of emerging memories (e.g., limited program cycles).

Disk-based file systems are not suitable for hybrid memory systems because NVMM has different characteristics from disks in both performance and consistency guarantees. As a result, naively running disk-based file systems on NVMM cannot fully exploit NVMM's high performance, and performance optimizations compromise consistency on system failure.

Existing NVMM file systems such as BPFS [3], PMFS [4], and ext4-DAX [10] also fail to provide the combination of performance and consistency NVMM should deliver. They use shadow paging and journaling to provide metadata atomicity, but these mechanisms incur high overheads that limit performance. PMFS and ext4-DAX avoid some of these costs by sacrificing data atomicity.

To provide consistency *and* high performance in an NVMM file system, we have created the *NOn-Volatile memory Accelerated (NOVA)* file system. NOVA rethinks conventional log-structured file system techniques to exploit the fast random access that hybrid memory systems provide. The result is that NOVA supports massive concurrency, keeps log sizes small, and minimizes garbage collection costs while providing strong consistency guarantees and very high performance.

Several aspects of NOVA set it apart from previous log-structured file systems. NOVA assigns each inode a separate log to maximize concurrency. NOVA stores the logs as linked lists, so they do not need to be contiguous in memory, and it uses atomic updates to a log's tail pointer to provide atomic log append. For operations that span multiple inodes, NOVA uses lightweight journaling.

NOVA uses copy-on-write for file data instead of storing it in the log. The resulting logs are compact, so the recovery process only needs to scan a small fraction of the NVMM. This also significantly reduces garbage collection overhead, allowing NOVA to sustain good performance even when the file system is nearly full. Finally, NOVA provides an `atomic mmap` interface that simplifies the tasks of writing programs that directly access NVMM via load and store instructions.

Our experiments show that NOVA is significantly faster than existing file systems in a wide range of applications and outperforms file systems that provide the same data consistency guarantees by 3.1x–13.5x. And when measuring garbage collection and recovery overheads, we find that NOVA provides stable performance under high NVMM utilization levels and fast recovery in case of system failure.

More details and results are available in our FAST '16 paper [11]. NOVA is open source and available at https://github.com/NVSL/NOVA.

NOVA Design Overview

NOVA is a log-structured POSIX file system. However, since it targets a different storage technology, NOVA looks very different from conventional log-structured file systems [9] that are built to maximize disk bandwidth.

We designed NOVA based on three observations. First, since NVMMs support fast, highly concurrent random accesses, using multiple logs does not negatively impact performance. Second, logs do not need to be contiguous, because random access is cheap. Third, data structures that support fast search (e.g., tree structures) are more difficult to implement correctly and efficiently in NVMM than in DRAM. Based on these observations, we made the following design decisions in NOVA.

Give each inode its own log: Unlike conventional log-structured file systems, each inode in NOVA has its own log, allowing concurrent updates across files without synchronization. It also means recovery is very fast, since NOVA can replay many logs simultaneously.

Keep logs in NVMM and indexes in DRAM: NOVA keeps log and file data in NVMM and builds highly optimized radix trees in DRAM to quickly locate file data. This means that searching the in-NVMM data structures is not usually necessary, so they can remain simple, easy to verify, and compact.

Implement the log as a singly linked list: The locality benefits of sequential logs are less important in NVMM-based storage, so NOVA uses a linked list of 4 KB NVMM pages to hold the log and stores the next page pointer in the end of each log page. As a result, NOVA can perform log cleaning at fine-grained, page-size granularity, and reclaiming log pages that contain only stale entries requires just a few pointer assignments.

Use logging to provide atomicity for simple, common-case operations: NOVA is log-structured because this provides cheaper atomicity for simple updates than journaling or shadow paging. To atomically write data to a log, NOVA first appends data to the log and then atomically updates the log tail to commit the updates, thus avoiding both the duplicate writes of journaling and the cascading updates of shadow paging.

Use lightweight journaling for more complex operations: Some directory operations, such as a move between directories, span multiple inodes. For these, NOVA uses journaling to atomically update multiple logs: NOVA first writes data at the end of each inode's log, and then journals the log tail updates to update them atomically. NOVA journaling is lightweight since it only involves log tails (as opposed to file data or metadata). The journals are very small—less than 64 bytes—since the most complex POSIX operation (`rename()`) involves up to four inodes, and each journal entry consists of eight bytes for the address of the log tail pointer and eight bytes for the updated value.

Use shadow paging for file data: NOVA uses copy-on-write for modified file data and appends metadata for the write to the log. The metadata describe the update and point to the data pages.

Using copy-on-write for file data results in shorter logs, accelerating the recovery process. It also makes garbage collection simpler and more efficient, since NOVA never has to copy file data out of the log to reclaim a log page. Finally, since it can reclaim stale data pages immediately, NOVA can sustain performance even under heavy write loads and high NVMM utilization levels.

Implementing NOVA

NOVA's core data structures focus on making file and directory operations fast and efficient. They also provide support for an mmap interface that makes using raw NVMM easier for programmers, while providing for efficient garbage collection and fast recovery in the case of a system failure.

NVMM Data Structures and Space Management

Figure 1 shows the high-level layout of NOVA data structures in a region of NVMM that it manages. NOVA divides the NVMM into four parts: the superblock and recovery inode, the inode

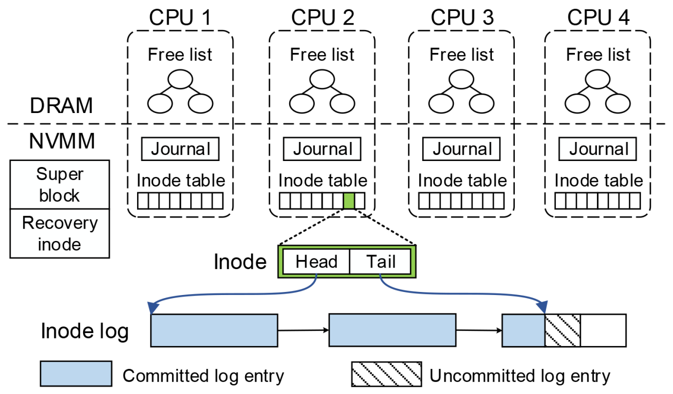


Figure 1: **NOVA data structure layout.** NOVA has per-CPU free lists, journals, and inode tables to ensure good scalability. Each inode has a separate log consisting of a singly linked list of 4 KB log pages; the tail pointer in the inode points to the latest committed entry in the log.

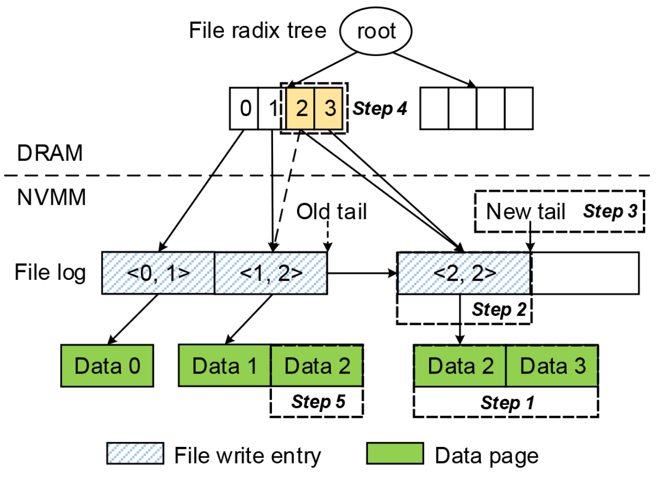


Figure 2: **NOVA file structure.** An 8 KB (i.e., two-page) write to page 2 (<2, 2>) of a file requires five steps. NOVA first writes a copy of the data to new pages (step 1) and appends the file write entry (step 2). Then it updates the log tail (step 3) and the radix tree (step 4). Finally, NOVA returns the old version of the data to the allocator (step 5).

tables, the journals, and log/data pages. The superblock contains global file system information, the recovery inode stores recovery information that accelerates NOVA remount after a clean shutdown, the inode tables contain inodes, the journals provide atomicity to directory operations, and the remaining area contains NVMM log and data pages.

To ensure high scalability, NOVA maintains an inode table, journal, and NVMM free page list at each CPU to avoid global locking and scalability bottlenecks: partitioning the inode table across CPUs avoids inode allocation contention and allows for parallel scanning in failure recovery. Per-CPU journals allow for concurrent transactions, and per-CPU NVMM free list provides concurrent NVMM allocation and deallocation. NOVA puts the free lists in DRAM to reduce consistency overheads.

Figure 2 shows the structure of a NOVA file. A NOVA inode contains pointers to the head and tail of its log. The log is a linked list of 4 KB pages, and the tail always points to the latest committed log entry. A file inode's log contains two kinds of log entries: inode update entries for metadata modifications and file write entries for writes. Each open file has a radix tree in DRAM to locate data in the file by the file offset. NOVA scans the log from head to tail to rebuild the DRAM data structures when the system accesses the inode for the first time.

NOVA provides fast atomicity for metadata, data, and mmap updates using a technique that combines log structuring and journaling. This technique uses three mechanisms:

64-bit atomic updates: NOVA uses 64-bit in-place writes to directly modify metadata for some operations (e.g., the file's `atime` for reads) and uses them to commit updates to the log by updating the inode's log tail pointer.

Logging: NOVA uses the inode's log to record operations that modify a single inode. These include operations such as `write`, `msync`, and `chmod`. The logs are independent of one another.

Lightweight journaling: For directory operations that require changes to multiple inodes (e.g., `create`, `unlink`, and `rename`), NOVA uses lightweight journaling to provide atomicity.

File Operations

NOVA uses copy-on-write for file data. On each write, NOVA writes the new version of the modified pages to newly allocated NVMM. Then it appends a write entry to the inode's log that describes the write and points to those pages.

Figure 2 illustrates a `write` operation. The notation <*file page offset, number of pages*> denotes the page offset and number of pages a `write` affects. The first two entries in the log describe two `writes`, <0, 1> and <1, 2>, of 4 KB and 8 KB (i.e., one and two pages), respectively. A third, 8 KB `write`, <2, 2>, is in flight.

To perform the <2, 2> `write`, NOVA fills data pages and then appends the <2, 2> entry to the file's inode log. Then NOVA atomically updates the log tail to commit the write, and updates the radix tree in DRAM, so that offset "2" points to the new entry. The NVMM page that holds the old contents of page 2 returns to the free list immediately. During the operation, a per-inode lock protects the log and the radix tree from concurrent updates. When the `write` system call returns, all the updates are persistent in NVMM.

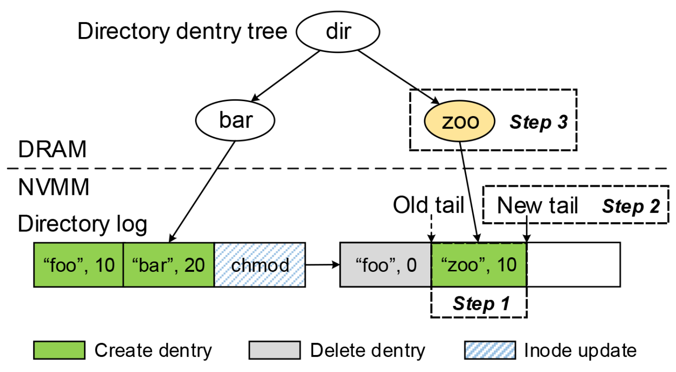


Figure 3: **NOVA directory structure.** Dentry is shown in <name, inode_number> format. To create a file, NOVA first appends the dentry to the directory's log (step 1), updates the log tail as part of a transaction (step 2), and updates the radix tree (step 3).

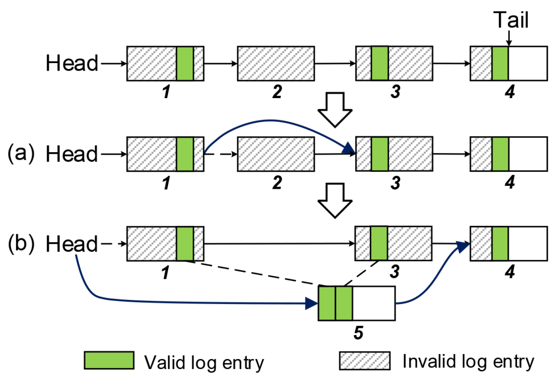


Figure 4: **NOVA log cleaning.** The linked list structure of log provides simple and efficient garbage collection. Fast GC reclaims invalid log pages by deleting them from the linked list (a), while thorough GC copies live log entries to a new version of the log (b).

If NOVA cannot find a contiguous region of NVMM big enough for the write, it will require multiple log entries. In this case NOVA breaks the `write` into multiple write entries and appends them all to the log to satisfy the request. To maintain atomicity, NOVA commits all the entries with a single update to the log tail pointer.

Directory Operations

Each directory inode's log holds two kinds of entries: directory entries (dentry) and inode update entries. Dentries include the name of the child file/directory, its inode number, and timestamp. NOVA uses the timestamp to atomically update the directory inode's *mtime* and *ctime* with the operation. NOVA appends a dentry to the log when it creates, deletes, or renames a file or subdirectory under that directory.

NOVA adds inode update entries to the directory's log to record updates to the directory's inode (e.g., for `chmod` and `chown`). These operations modify multiple fields of the inode, and the inode update entry provides atomicity.

Figure 3 illustrates the creation of file *zoo* in a directory that already contains file *bar*. The directory has recently undergone a `chmod` operation and used to contain another file, *foo*. The log entries for those operations are visible in the figure. NOVA first selects and initializes an unused inode in the inode table for *zoo* and appends a create dentry of *zoo* to the directory's log. Then NOVA uses the current CPU's journal to atomically update the directory's log tail and set the valid bit of the new inode. Finally, NOVA adds the file to the directory's radix tree in DRAM.

Atomic-mmap

NOVA includes a novel direct NVMM access model with stronger consistency called `atomic-mmap`. When an application uses `atomic-mmap` to map a file into its address space, NOVA allocates *replica pages* from NVMM, copies the file data to the replica pages, and then maps the replicas into the address space. When the application calls `msync` on the replica pages, NOVA handles it as a `write` request described in the previous section, uses `movntq` operation to copy the data from replica pages to data pages directly, and commits the changes atomically. Comparing to PMFS and ext4-DAX that map the NVMM file data pages directly into the application's address space, `atomic-mmap` has higher overhead but provides stronger consistency guarantee.

Garbage Collection

NOVA uses two complementary techniques to reclaim dead log entries: *fast GC* and *thorough GC*. Both use the same criteria to determine whether a log entry is dead: namely, if it is not the last entry in the log and any of the following conditions are met:

- A file write entry is dead if it does not refer to valid data pages.
- An inode update that modifies metadata (e.g., *mode* or *mtime*) is dead if a later inode update modifies the same piece of metadata.
- A create dentry is dead if a corresponding delete dentry is appended to the log. Both dentries become invalid in this case.

Fast GC applies these rules to reclaim log pages that do not contain any live entries by deleting the page from the log's linked list. Figure 4a illustrates this: originally, the log has four pages and all the entries in page 2 are dead. NOVA atomically updates the next page pointer of page 1 to point to page 3, freeing page 2.

Thorough GC compacts log pages by copying live data from several pages into a new page. Figure 4b illustrates thorough GC after fast GC is complete. NOVA allocates a new log page 5 and copies valid log entries in pages 1 and 3 into it. Then NOVA links page 5 to page 4 to create a new log and replace the old one. NOVA does not copy the live entries in page 4 to avoid updating the log tail, so that NOVA can atomically replace the old log by updating the log head pointer.

Workload	Average file size	I/O size (r/w)	Threads	R/W ratio	Number of files (Small/Large)
Fileserver	128 KB	16 KB/16 KB	50	1:2	100K:400K
Webproxy	32 KB	1 MB/16 KB	50	5:1	100K:1M
Webserver	64 KB	1 MB/8 KB	50	10:1	100K:500K
Varmail	32 KB	1 MB/16 KB	50	1:1	100K:1M

Table 1: Filebench workload characteristics. The selected four workloads have different read/write ratios and access patterns.

Mounting, Unmounting, and Recovery

NOVA's design allows for fast mounting and unmounting as well as efficient recovery after a system failure. When NOVA mounts a file system, it must construct two kinds of data structure in DRAM: the per-inode radix trees and the NVMM allocator.

After either a clean unmount or system failure, NOVA rebuilds the radix trees on demand when any application opens a file or directory.

Reconstructing the NVMM allocator state, however, cannot wait. During a normal unmount, NOVA writes the NVMM page allocator state in the recovery inode's log. Then it can quickly rebuild the allocator on remount.

After a system failure, NOVA rebuilds the NVMM allocator by scanning all the inode logs. Fortunately, NOVA can use multiple threads to perform the scan in parallel, and the logs are small since they only contain metadata.

Evaluation

NOVA aims to provide strong consistency, high performance, and fast recovery, and our results show that it achieves these goals.

We have implemented NOVA in the Linux kernel version 4.0 using the existing NVMM hooks that the kernel provides. It currently passes the Linux POSIX file system test suite [7].

We measure NOVA's performance using Intel's Persistent Memory Emulation Platform (PMEP) [4], a dual-socket Xeon platform with special CPU microcode and firmware that allows it to emulate some aspects of NVMM performance with DRAM. PMEP supports configurable latencies and bandwidth for the emulated NVMM, allowing us to explore NOVA's performance on a variety of future memory technologies. PMEP emulates `clflushopt` (efficient cache line flush), `clwb` (cache line write back), and `PCOMMIT` (persistent commit) instructions with processor microcode.

In our tests we configure the PMEP with 32 GB of DRAM and 64 GB of NVMM. We choose two configurations for PMEP to emulate different NVMM technologies: for STT-RAM we use the same read latency and bandwidth as DRAM, and configure `PCOMMIT` to take 200 ns (to match projections for STT-RAM write times); for PCM we use 300 ns for the read latency and reduce the write bandwidth to 1/8 of DRAM while increasing `PCOMMIT` time to 500 ns. `clwb` takes 40 ns in both configurations.

We compare NOVA to seven other file systems: Two NVMM file systems, PMFS and ext4-DAX, are journaling file systems. Two others, NILFS2 and F2FS, are log-structured file systems. We also compare to ext4 in default mode (ext4) and in data journal mode (ext4-data), which provides data atomicity. Finally, we compare to btrfs, a copy-on-write Linux file system. PMFS, ext4-DAX, and NOVA are *Direct Access (DAX)* file systems that bypass the operating system page cache and access NVMM directly. Btrfs and ext4-data are the only two file systems in the group that provide the same strong consistency guarantees as NOVA. Ext4-DAX does not currently provide a data journaling option. We add `clwb` and `PCOMMIT` instructions to flush data where necessary in each file system, and use Intel persistent memory driver [6] to emulate an NVMM-based RAMDisk-like device.

Macrobenchmarks

We select four Filebench [5] workloads to evaluate the application-level performance of NOVA (Table 1). For each workload we test two data set sizes by changing the number of files. The *small* data set will fit entirely in DRAM, allowing file systems that use the DRAM page cache to cache the entire data set. The *large* data set is too large to fit in DRAM, so the page cache is less useful.

Figure 5 shows the Filebench throughput. In the fileserver workload, NOVA outperforms other file systems by 1.8x–16.6x on STT-RAM, and between 22% and 9.1x on PCM for the largest data set. NOVA outperforms ext4-data by 11.4x and btrfs by 13.5x on STT-RAM, while providing equivalent consistency guarantees. NOVA on STT-RAM delivers twice the throughput compared to PCM, because of PCM's lower write bandwidth.

Web proxy is a read-intensive workload, and it puts all the test files in one large directory. For the small data set, NOVA performs similarly to ext4 and ext4-DAX, and 2.1x faster than ext4-data. For the large workload, NOVA performs 36%–53% better than F2FS and ext4-DAX. PMFS and NILFS2 perform poorly in this test because their directory designs are not scalable.

Web server is a read-dominated workload and does not involve any directory operations. As a result, non-DAX file systems benefit significantly from the DRAM page cache, and the workload

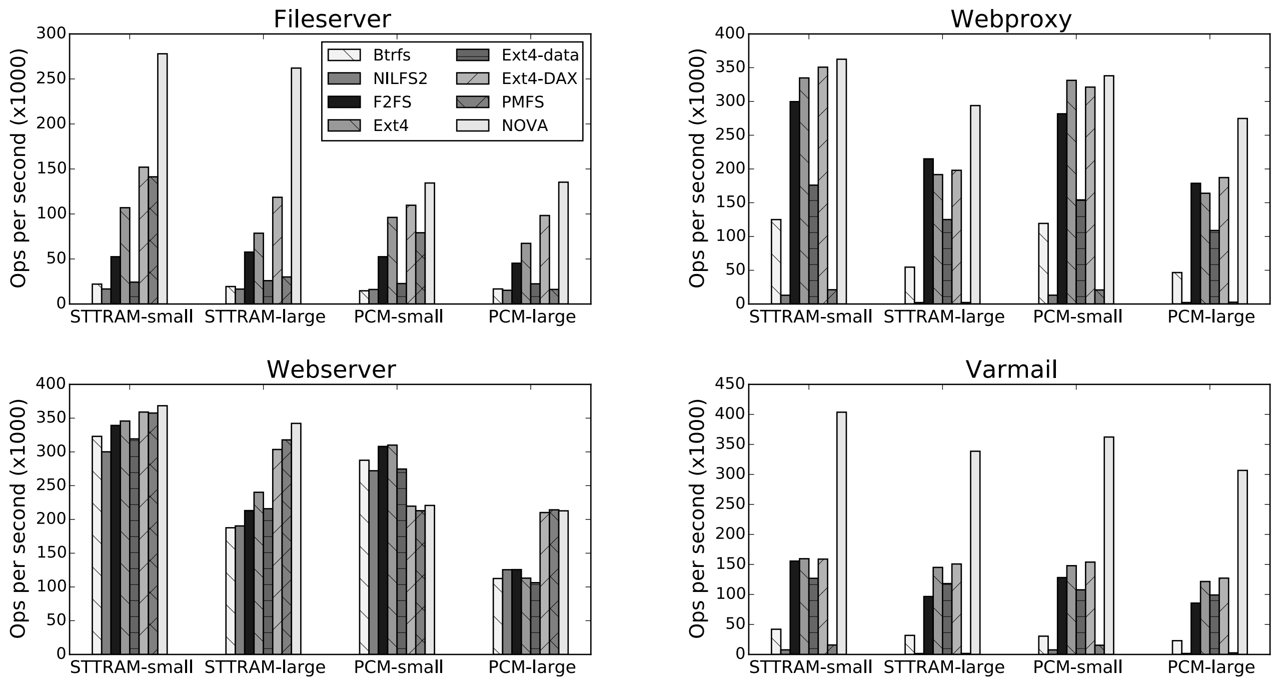


Figure 5: **Filebench throughput with different file system patterns and dataset sizes on STT-RAM and PCM.** Each workload has two data-set sizes so that the small one can fit in DRAM entirely while the large one cannot.

size has a large impact on performance. On STT-RAM, with the large data set, NOVA performs 63% better on average than non-DAX file systems. On PCM, for the small data set, non-DAX file systems are 33% faster on average due to DRAM caching. However, for the large data set, NOVA's performance remains stable while non-DAX performance drops by 60%.

Varmail emulates an email server with a large number of small files and involves both `read` and `write` operations. NOVA outperforms btrfs by 11.1x and ext4-data by 3.1x on average, and outperforms the other file systems by 2.2x–216x. NILFS2 and PMFS still suffer from poor directory operation performance.

Overall, NOVA achieves the best performance in almost all cases and provides data consistency guarantees that are as strong or stronger than the other file systems. The performance advantages of NOVA are largest on write-intensive workloads with large number of files.

Garbage Collection Efficiency

We designed NOVA to perform garbage collection efficiently and maintain stable performance under heavy write loads, even when the file system is nearly full. To verify these characteristics, we ran a 30 GB write-intensive fileserver workload and adjusted the amount of NVMM available to bring utilization to 95%. Then we compared NOVA's behavior with the other log-structured file systems, NILFS2 and F2FS. We ran the test with PMEP configured to emulate STT-RAM.

Figure 6 shows the result. NILFS2 failed after less than 10 seconds since it ran out of space due to garbage collection inefficiencies. F2FS failed after running for 158 seconds after suffering a 60% drop in throughput due to log cleaning overhead. NOVA outperformed F2FS by 12x, and the throughput remained stable over time.

We found that the longer NOVA runs, the more efficient fast GC becomes, eventually accounting for the majority of reclaimed pages.

Recovery Overhead

NOVA provides fast recovery from both normal dismounts and power failures. To measure the recovery overhead, we used the three workloads in Table 2. Each workload represents a different use case for the file systems: Videoserver contains a few large files accessed with large-size requests; mailserver includes a large number of small files and the request size is small; fileserver is in-between. For each workload, we measure the cost of mounting after a normal shutdown and after a power failure.

Table 3 summarizes the results. With a normal shutdown, NOVA recovers the file system in 1.2 ms, since NOVA can restore the allocator state from the a checkpoint. After a power failure, NOVA recovery time increases with the number of inodes and as the I/O operations that created the files become smaller (since small I/O operations result in more log entries). Recovery runs faster on STT-RAM than on PCM because PCM has higher read latency. On both PCM and STT-RAM, NOVA is able to recover 50

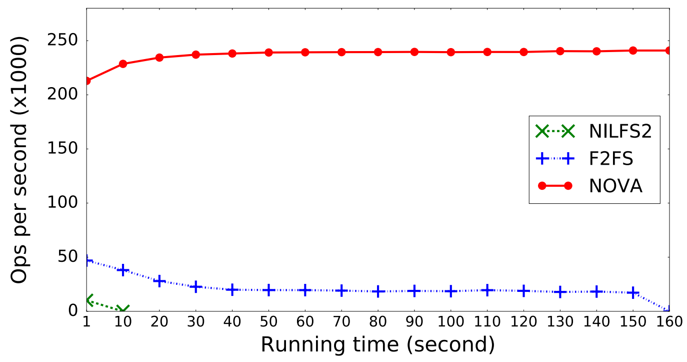


Figure 6: **Performance of a full file system.** The test runs a 30 GB fileserver workload under 95% NVMM utilization with different durations.

GB data in 116 ms, achieving failure recovery bandwidth higher than 400 GB/s.

Conclusion

We have implemented and described NOVA, a log-structured file system designed for hybrid volatile/non-volatile main memories. NOVA extends ideas of LFS to leverage NVMM, yielding a simpler, high-performance file system that supports fast and efficient garbage collection and quick recovery from system failures. Our measurements show that NOVA outperforms existing NVMM file systems by a wide margin on a wide range of applications while providing stronger consistency and atomicity guarantees.

Data set	File size	Number of files	Data-set size	I/O size
Videoserver	128 MB	400	50 GB	1 MB
Fileserver	1 MB	50,000	50 GB	64 KB
Mailserver	128 KB	400,000	50 GB	16 KB

Table 2: **Recovery workload characteristics.** The number of files and typical I/O size both affect NOVA's recovery performance.

Data set	Videoserver	Fileserver	Mailserver
STTRAM-normal	156 μs	313 μs	918 μs
PCM-normal	311 μs	660 μs	1197 μs
STTRAM-failure	37 ms	39 ms	72 ms
PCM-failure	43 ms	50 ms	116 ms

Table 3: **NOVA recovery time on different scenarios.** NOVA is able to recover 50 GB data in 116 ms in case of power failure.

Acknowledgments

This work was supported by STARnet, a Semiconductor Research Corporation program, sponsored by MARCO and DARPA. We would like to thank John Ousterhout, Niraj Tolia, Isabella Furth, Rik Farrow, and the anonymous FAST reviewers for their insightful comments and suggestions. We are also thankful to Subramanya R. Dulloor from Intel for his support and hardware access.

References

[1] Intel and Micron produce breakthrough memory technology: http://newsroom.intel.com/community/intel_newsroom/blog/2015/07/28/intel-and-micron-produce-breakthrough memory-technology.

[2] A. Akel, A. M. Caulfield, T. I. Mollov, R. K. Gupta, and S. Swanson, "Onyx: A Prototype Phase Change Memory Storage Array," in *Proceedings of the 3rd USENIX Conference on Hot Topics in Storage and File Systems (HotStorage '11)*.

[3] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee, "Better I/O through Byte-Addressable, Persistent Memory," in *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP '09)*, pp. 133–146.

[4] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson, "System Software for Persistent Memory," in *Proceedings of the 9th European Conference on Computer Systems (EuroSys '14)*, ACM, pp. 15:1–15:15.

[5] Filebench file system benchmark: http://sourceforge.net/projects/filebench.

[6] PMEM: the persistent memory driver + ext4 direct access (DAX): https://github.com/01org/prd.

[7] Linux POSIX file system test suite: https://lwn.net/Articles/276617/.

[8] S. Raoux, G. Burr, M. Breitwisch, C. Rettner, Y. Chen, R. Shelby, M. Salinga, D. Krebs, S.-H. Chen, H. L. Lung, and C. Lam, "Phase-Change Random Access Memory: A Scalable Technology," *IBM Journal of Research and Development,* vol. 52, no. 4.5, July 2008, pp. 465–479.

[9] M. Rosenblum and J. K. Ousterhout, "The Design and Implementation of a Log-Structured File System," *ACM Transactions on Computer Systems (TOCS)*, vol. 10, no. 1, 1992, pp. 26–52.

[10] M. Wilcox, "Add Support for NV-DIMMs to ext4: https://lwn.net/Articles/613384/.

[11] J. Xu and S. Swanson, "NOVA: A Log-Structured File System for Hybrid Volatile/Non-Volatile Main Memories," in *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST '16)*, pp. 323–338.

[12] P. Zhou, B. Zhao, J. Yang, and Y. Zhang, "A Durable and Energy Efficient Main Memory Using Phase Change Memory Technology," in *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA '09)*, ACM, pp. 14–23.

Interview with Timothy Roscoe

RIK FARROW

Timothy Roscoe (aka Mothy) is a full Professor in the Systems Group of the Computer Science Department at ETH Zurich. He received a PhD from the Computer Laboratory of the University of Cambridge, where he was a principal designer and builder of the Nemesis operating system, as well as working on the Wanda microkernel and Pandora multimedia system. After three years working on Web-based collaboration systems at a startup company in North Carolina, Mothy joined Sprint's Advanced Technology Lab in Burlingame, California, working on cloud computing and network monitoring. He then joined Intel Research at the University of California, Berkeley in April 2002, as a principal architect of PlanetLab, an open, shared platform for developing and deploying planetary-scale services. In September 2006, he spent four months as a visiting researcher in the Embedded and Real-Time Operating Systems group at National ICT Australia in Sydney before joining ETH Zurich in January 2007. His current research interests include network architecture and the Barrelfish multicore research operating system. He was recently elected Fellow of the ACM for contributions to operating systems and networking research.

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

I interviewed Mothy Roscoe over six years ago, after I became aware of the Barrelfish operating system [1]. Barrelfish was designed specifically for the new generation of many-core CPU chips, and Mothy was one of the people on the team who built the OS [2].

Since that time, Intel released the Single-chip Cloud Computer [3], and the Barrelfish team created a port for this unusual chip. The SCC was designed as a research microprocessor with 48 cores and a message passing system. Today, Intel has moved toward the Xeon Phi, a chip that sits on the PCI express bus, exposes many processing cores, and provides a Linux-based stack for communicating with the Phi. The Phi has 68 cores and is designed for HPC.

A number of sophisticated 64-bit ARM multicore server chips are appearing that look like a great fit for Barrelfish. Mothy told me that Barrelfish does support some of these, but actually getting documentation from vendors can prove difficult in some cases.

I'd had the opportunity to chat with Mothy during many systems conference luncheons, and I wanted to follow up on some of those conversations and share them.

Mothy is co-chair of OSDI '16, along with Kim Keeton of HP Labs and HP Enterprise. The discussion of systems conference program committees that I mention in "Musings" (page 2) occurred after we had conducted this interview.

Rik: Please update us on what has been happening with Barrelfish since we last talked (April 2010).

Mothy: Quite a lot—it's been an interesting journey! To some extent the focus of the project has changed, but it's remarkable how many of our original goals, and our conjectures about the challenges of future hardware, turned out to be on the mark.

When we started in 2007, we thought that the hard problems in OS design were scaling, heterogeneity, and diversity. Those challenges led us to the multikernel model: by structuring the OS as a distributed set of cores communicating with messages, we handle heterogeneity, lack of cache coherence and/or shared memory, dynamic cores, etc. in a single elegant framework.

The one challenge the multikernel model does not solve directly is scalability. However, we find it a lot easier to think about scaling in the context of message passing rather than shared memory—and you also see this trend in other areas like HPC and large-scale "Big Data" applications as well.

We've learned a lot about modern hardware so far in building Barrelfish and a lot about how to write a modern OS for that hardware. There is a huge difference between building a real OS from scratch with a different design versus tweaking existing Linux or Windows kernels. We were lucky to have the opportunity to pull off a sufficiently large engineering project over a long period of time. For us, the big payoff happened after six years on when companies started to get very interested in Barrelfish—that's a long time by the standards of most university research projects.

Along the way, we've published a lot of papers and graduated a bunch of PhD students. Academically, it's been a huge success so far. A common criticism of big projects like this is that they don't generate enough papers in the modern academic climate—several colleagues (not at ETH) have suggested that it's better to focus on smaller projects and higher paper counts so that students find it easier to get academic jobs.

I want the students to publish, but I feel strongly that paper mills are not the only way to run a university research group, and paper-driven research isn't a good way to have long-term impact (it's also not ETH's mission). Simon Peter, the PhD student who wrote the first line of code of Barrelfish, is now a professor at University of Texas, Austin. He also won Best Paper at OSDI 2014 with Arrakis [5], a variant of Barrelfish, so we seem to be doing OK nine years on.

In building Barrelfish, we also made a bunch of decisions that were somewhat arbitrary at the time—we really felt they were a good way to build an OS, but we didn't view them as essential to the research. In retrospect, however, many of these choices were highly fortuitous.

For example, we adopted a capability model (which we extended from seL4) for managing all memory resources, and this turned out to have profound consequences much later. It's only now that we're finding this is a great match for very large main memories, like HP's "The Machine" project.

The big change came in about 2012. Barrelfish had become a useful vehicle for OS research and teaching at ETH, despite the sometimes uphill struggle to convince people that "it's not Linux" was not a showstopper for a realistic OS, even in the research community.

At this point, we started being contacted by companies (hardware vendors and others) who had become interested in Barrelfish. They were building hardware that didn't look like a 1980s VAX, and it was dawning on them that Linux wasn't a good fit for these new hardware designs.

It was this interest that convinced us to keep going with the project rather than move on to new things. It was great to feel that the ideas that Paul Barham, Rebecca Isaacs, Simon Peter, Andrew Baumann, myself, and others had started with back in 2007 had finally been vindicated.

We've now got a great core development group at ETH (including one full-time software engineer and, hopefully, more in the future), we accept external contributions (both patches and pull requests on GitHub), and we're looking to support more hardware as we work with more vendors.

We've also got a ton of ideas, thoughts, war stories, etc. that we'd really like to talk about, but which are a bit of a challenge to fit into traditional computer science conference publishing or system documentation—we're starting a blog of these to see what interest there is out there.

Rik: What's the main research direction of the project now?

Mothy: One of the biggest driving challenges right now is hardware complexity. Modern computer hardware is incredibly complex in terms of peripheral devices, interconnects, memory hierarchies, etc.

To take one example: most people think they know what a physical address is; every memory cell or device register sits at a unique physical address, which is what you can use to access that location by having the MMU put out that address on the memory interconnect. Unfortunately, this just isn't true any more.

Instead, within a single machine (even a single SoC), different cores will see the same memory location appear at different physical addresses. Each core will only be able to address a subset of all the memory locations (storage cells or device registers), and these subsets are different, but not necessarily disjoint: they intersect in interesting ways. The access latency and bandwidth to a given location also varies, of course. For any pair of cores in the machine, the same location might be at the same address, or different addresses, or only addressable from one core, and might or might not be coherent in cache.

This happens on modern PCs, phone SoCs, and pretty much any other piece of "mainstream" hardware. We also see analogous complexity and diversity across systems in DMA engines, interrupt routing, network interfaces, and so on.

So what do we do? Nobody has a really good, crisp, formal description of what, say, a physical address is these days. The closest you find in traditional OS designs is a device tree (http://www.devicetree.org/), but device trees are really a file format—it doesn't capture semantics in a way you can make strong statements about.

We'd like to be able to put the hardware/software interface that OSes must use on a much more sound formal footing. What we're doing is writing semi-formal descriptions of all the memory systems, interrupt routing models, interconnect topologies that we can find, and then devising representations of these in subsets of first-order logic.

The short-term benefit of this is that Barrelfish can easily adapt to new hardware online—we've always programmed PCIe buses using Constraint Logic Programming in Barrelfish at boot time, and we're doing a lot more in this line. It's just a much easier way to engineer a more portable, general purpose OS for modern hardware. And in the medium term, of course, we can use these models and representations to provide a basis for formal verification of system software.

The ultimate goal, however, is a "software-centric" description of hardware that allows us as OS designers to talk about what kinds of hardware designs are "tasteful" as far as the OS is concerned, and what hardware design patterns are not. OS people are pretty good at critiquing bad hardware designs (and there are many!), but hardware designers pretty much ignore OS folks, and one reason for this is that we're not nearly as good at saying up front what the rules and patterns are for good hardware interface design.

This sounds philosophical, but there's very little academic work on this, and it's amazing how much traction we've got for these long-term ideas from industry partners. It's fundamental work that also has direct short-term applicability in industry.

Rik: Any plans on porting Barrelfish to more ARM-based server SoCs? It sounds like some of them might be vaguely similar to the Intel cloud-on-a-chip.

Mothy: We're always interested in future hardware platforms, if we can find out anything about them :-). If you're writing an operating system for new hardware in an academic environment, a constant headache is getting documentation. Pretty much any hardware vendor assumes you want to run Linux on their chip, and that's it, so why would you want documentation?

We sign NDAs, and sometimes this really helps, and sometimes it doesn't. But even finding someone to talk to about getting documentation is often the challenge.

Sometimes it works out great—Intel actually approached us before the Single-chip Cloud Computer was announced some years back, and my student did a port to the SCC that was ready when they launched. ARM, Intel, and some vendors (such as TI) released extremely detailed documentation. AMD usually does as well, but we've found nothing about the Seattle [A1100] processors, for example, and haven't found anyone to ask about it either.

There is a flurry of really interesting ARMv8-based server chips appearing, however, and we're excited about supporting them. We currently support AppliedMicro's X-Gene 1 SoC and ARM's FAST models for emulated hardware, but we'll talk to anyone who is prepared to share documentation with us.

Rik: In conversations we've had during USENIX conferences, you mentioned that Barrelfish will only support certain programming languages. Can you explain the thinking behind that decision?

Mothy: Actually, we're happy if Barrelfish supports any programming language that people would like. As a small team, there's a limit to the number that we can support ourselves, but we're happy to accept contributions!

We've had various student projects porting language runtimes to Barrelfish, and it's usually not too much of a problem. The key challenge is generally that languages often implicitly assume a POSIX-like system, and Barrelfish deliberately isn't like POSIX. This makes a fast Java runtime, for example, more work than Rust, which was extremely easy to bring up.

What you're probably referring to is the group decision about which languages we could use in the OS itself. Remember this was back in 2008, so this is before Go, D, Rust, Swift, and Dart had traction. The discussion was remarkably short and pretty much unanimous: we decided on C, assembly, Haskell, and Python for tools. We also felt happy with OCaml and Ruby, but in the event we didn't use them. We unanimously banned Java, C#, C++, and Perl. Nothing has happened to make us regret this decision.

Rik: Functional programming languages are becoming increasing popular. We now have a unikernel OS, MirageOS, but it relies on OCaml programming. When I mentioned just how hard I found it to write functional programs, you suggested that it might take someone six months to switch over to a functional programming style. Could you elaborate?

Mothy: We don't have any studies to back this up, and it depends ultimately on the programmer. However, OS kernels have always employed a variety of programming paradigms expressed in assembly or C, and good OS kernel hackers are generally comfortable with a variety of different ways of expressing computations inside the OS. You see a lot of snippets of functional programming in the Linux kernel, for example.

Choice of languages are a different matter. It's useful to contrast unikernels from operating systems: they're very different, and ultimately complementary.

We like to use a "functional" (in a different sense) definition of an OS—it is "that which manages, multiplexes, and protects the hardware resources of the machine." In this sense, it could be a hypervisor like Xen, or a traditional OS like Linux or Windows, or a multikernel like Barrelfish.

A unikernel like MirageOS is essentially a runtime for a single application that executes in a resource container. Some people call this a LibraryOS—a term which goes back to exokernel systems like Aegis and Nemesis.

For an OS, a garbage-collected language is problematic because you are interested in providing performance isolation between competing, untrusted applications (such as containers). Hence, implementations using a language with more predictable performance like C, Rust, or even C++ make a lot of sense.

For a unikernel, you're not worried about scheduling, resource sharing, or crosstalk since you're already isolated in a container. What you really want here is something which works well from a software development and correctness perspective, and languages like OCaml are great for this kind of environment.

As an aside, if you move to a world where all your applications are running in containers over unikernels, the question naturally arises as to what the "ideal" underlying OS is. It's not Linux or Windows, and it's not Barrelfish either yet, but we are evolving Barrelfish that way—the Arrakis work is a step in that direction, for example.

References

[1] R. Farrow, "The Barrelfish Multikernel: Interview with Timothy Roscoe." *;login:*, vol. 35, no. 2 (April 2010): bit.ly/29lczBY.

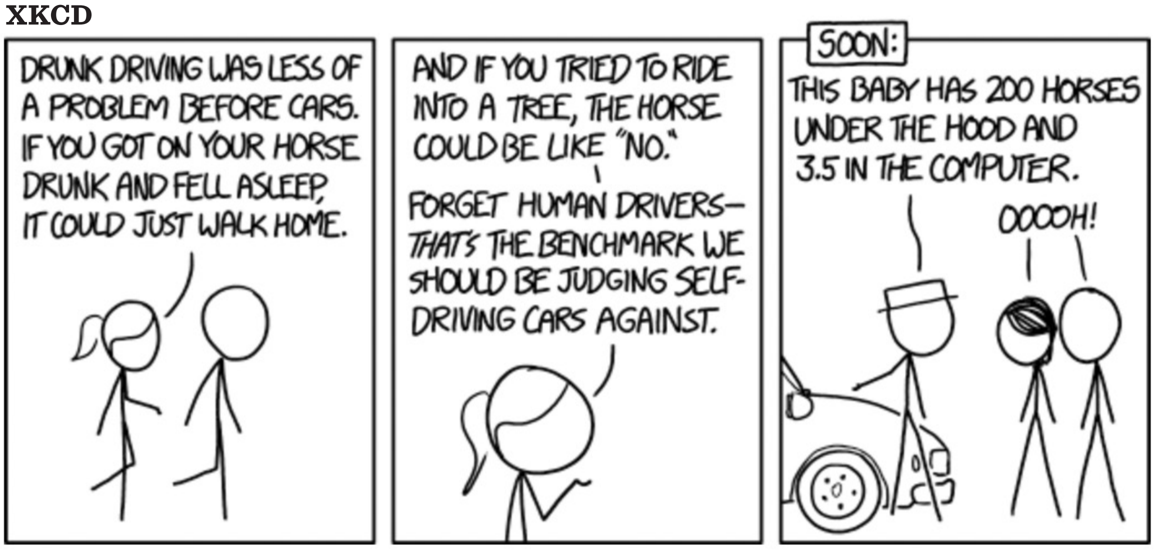
[2] A. Baumann, P. Barham, P. E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania, "The Multikernel: A New OS Architecture for Scalable Multicore Systems," in *Proceedings of the 22nd ACM Symposium on OS Principles,* October 2009: http://www.sigops.org/sosp/sosp09/papers/baumann-sosp09.pdf.

[3] "Intel Single-chip Cloud Computer," 2003: intel.ly/29oPSNx.

[4] Intel Xeon Phi: http://www.intel.com/content/www/us/en/processors/xeon/xeon-phi-detail.html.

[5] S. Peter and T. Anderson, "Arrakis: The Operating System as Control Plane." *;login:*, vol. 38, no. 4 (August 2013): bit.ly/29gu97f.

XKCD


DRUNK DRIVING WAS LESS OF A PROBLEM BEFORE CARS. IF YOU GOT ON YOUR HORSE DRUNK AND FELL ASLEEP, IT COULD JUST WALK HOME.
AND IF YOU TRIED TO RIDE INTO A TREE, THE HORSE COULD BE LIKE "NO."
FORGET HUMAN DRIVERS— THAT'S THE BENCHMARK WE SHOULD BE JUDGING SELF-DRIVING CARS AGAINST.
SOON:
THIS BABY HAS 200 HORSES UNDER THE HOOD AND 3.5 IN THE COMPUTER.
OOOOH!

xkcd.com

Runway
A New Tool for Distributed Systems Design

DIEGO ONGARO

Diego Ongaro is the creator of Runway and is a Lead Software Engineer on the Compute Infrastructure team at Salesforce. He is interested in improving the way people build distributed systems. He received his PhD in 2014 from Stanford University, where he worked on Raft, a consensus algorithm designed for understandability, and RAMCloud, a low-latency storage system.
dongaro@salesforce.com

We strive to build correct systems that are always on and always fast. They must be distributed, yet the complexity inherent in distributed systems poses a major design challenge. Runway is a new tool for distributed systems design, enabling interactive visualizations to help people learn about designs, and simulation and model checking to help evaluate their key properties. This article introduces Runway and discusses key issues in modeling distributed systems.

More than ever, companies are building and deploying distributed systems. They are forced to distribute computation and data across servers to improve the availability, performance, and scale of their services. Unfortunately, this comes at a steep cost of complexity:

- In a distributed system, multiple servers can operate concurrently. Their events can end up happening in orders that are hard to anticipate.
- Due to network latency, by the time a server receives a message, its contents may already be stale.
- Failures such as server crashes and network partitions are common at scale, and they can happen at any time, even while the system is trying to recover from another failure.
- Because servers are separated by a network, visibility into running systems is reduced, and debugging environments are limited.

The best way to manage this complexity is to focus more efforts on system design. In the design phase, we should communicate clearly about a design and also evaluate that design's key properties, such as its understandability and simplicity, correctness, availability, performance, and scalability. Exploring and resolving design issues early, before investing heavily in implementation, should help lower the cost of developing distributed systems and improve their quality.

Many existing tools aim to help with specifying, checking, or simulating distributed system models (some are listed on the Runway wiki [1]). However, none of these seems to be widely used for designing distributed systems in industry. Instead, industry engineers still rely on primitive tools like whiteboards, back-of-the-envelope calculations, and design documents. These are valuable, but they fall short of communicating clearly about a design or evaluating its important properties. Why don't industry engineers use more sophisticated design tools? We can only assume that they are unwilling, existing tools are impractical, or the engineers haven't found the right tools. If it's the former, there is little hope. But if it's the latter two, Runway might have a chance.

Runway is a new design tool for distributed systems. It's not technically superior to existing tools, but it may be better optimized for a chance at widespread adoption in industry. There are three primary reasons for this:

1. **Integration:** Runway combines specification, model checking, simulation, and visualization in the same tool. Integrating many components might tip the cost-benefit calculation in Runway's favor: you can write one system model and get a lot of value from it. Compared to using separate tools, Runway has only a single learning curve. Plus, you can start by specifying and visualizing a model, then decide how to evaluate it later (using model checking, simulation, or both).
2. **Usability:** Runway aims to be approachable, with only a small learning curve. The interactive visualizations allow people with no special knowledge of Runway to learn about a design. For modeling, Runway's specification language is designed to be familiar to most engineers and encourages simple code without many abstractions.
3. **Social:** Runway visualizations run in a Web browser, enabling people to share their models easily. We're currently designing a registry to help people discover other models, as well as a component system to enable using one model within another. We hope a community will grow around modeling systems in Runway and learning about them.

Although Runway is still early in its development, it can already provide significant value. A public instance of Runway is available at https://runway.systems/, and its source code [2] is freely available under the MIT license.

Overview of Runway

A Runway model consists of a specification and a view. The specification describes the model's state and how that state may change over time. Specifications are written in code using a new domain-specific language. This language aims to be familiar to programmers and have simple semantics, while expressing concurrency in a way suited for formal and informal reasoning. A specification describes a *labeled transition system*, which is like a state machine, for how state changes. It can also include *invariants*, properties that must hold for every correct state. The view draws a model's state visually. For example, the view for the Runway model of the Raft consensus algorithm [3] is shown in Figure 1.

Runway includes a compiler for its specification language, and it can execute the specification using a randomized simulator. This produces an *execution*, an ordered history or schedule of events that captures a sequence of state changes. Runway can then visualize or animate these state changes over time. Runway employs the model's view for the main component of the visualization and also adds several generic widgets, including a timeline, an editable table of the model's entire state, and a toolbar of transition rules that can be applied to the state. The visualization is interactive, allowing users to manipulate the state of a model and see how it reacts. It also serves as a great debugger when developing specifications.

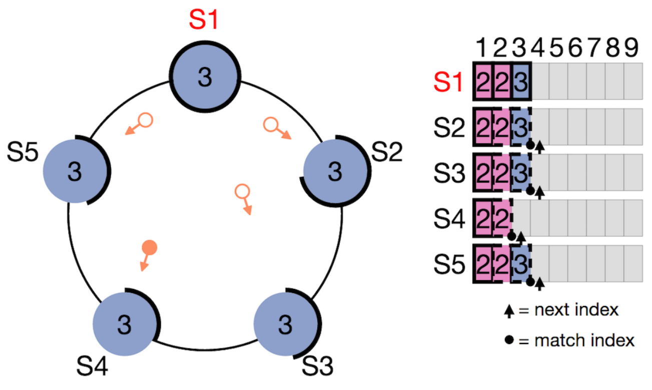


Figure 1: The view of the Raft model. The ring on the left is optimized for understanding leader election. Each server has a randomized election timer, drawn as an arc around the server. Heartbeat messages from the leader reset that timer, and a server begins an election when its timer expires. The servers' logs on the right are optimized for understanding log replication; they are lined up in tidy rows for easy comparisons. This interactive visualization is available at https://runway.systems/.

The simulator can do more than power a visualization: it can also collect data. A single execution can include many interesting data points, and as a planned feature, data could be aggregated across a family of executions. The data can be presented in the form of graphs, and by selecting a point on a graph, the visualization can load and replay the exact event of interest.

The final major component of Runway is the model checker, which can verify that a model will never break an invariant, up to some limit in model size. The model checker begins at the model's starting state and tries to explore all reachable states, evaluating the invariants at each step. It never expands the same state twice, using a hash table to track the states it's already visited. Runway's model checker today is quite slow; we plan to either implement optimizations from the literature or to have Runway invoke an efficient model checker behind-the-scenes. If the model checker finds a bug, it can produce an execution showing how the model reaches a bad state. As a planned feature, this execution could be loaded into the visualization so that a user could easily understand what went wrong.

Runway's Specification Language

Although Runway's specification language is still a work in progress, several basic principles are shaping its design:

- It aims to be easy for industry developers to read and write, with only a small learning curve. Although a functional approach is possible, an imperative, procedural approach is likely to be more familiar.
- It intentionally includes a limited set of language features, favoring specifications with straightforward code, even at the cost of larger specification sizes.

- Its strong type system is intended to help developers avoid silly errors like typos and misused variables.
- It permits modeling concurrency without writing concurrent code. Transition rules are applied atomically, one at a time. To model concurrency, one rule can model the start of a long-lived, concurrent operation, and another can model its completion.
- It keeps all state explicitly in global variables, which simplifies reasoning about the current state. This is in contrast with general-purpose languages, which use the instruction pointer to track information, without giving it a name.
- Although Runway does not yet include an efficient implementation, it must be possible to evaluate Runway models efficiently, and, especially for model checking, their state must be efficient to copy and hash. We hope to store the global state variables contiguously in memory even as the language evolves, although this may need to be relaxed in favor of more flexible models (every variable has a static upper bound on its size today).

The Too Many Bananas problem serves as a good example to illustrate Runway's specification language. It is a simple concurrency problem similar to those taught in introductory systems classes. You live in a house with roommates, and everyone likes to eat bananas. When you run out of bananas, you go to the store to buy more and bring those home. Due to a race condition, it's possible for your roommate to leave for the store while you're already out buying more bananas. When you both return home, you might end up with too many bananas, a critical problem since bananas spoil over time.

This specification models the Too Many Bananas problem:

```
01 var bananas : 0..100;
02 type Person : either {
03   Happy,
04   Hungry,
05   GoingToStore,
06   ReturningFromStore {
07     carrying: 0..8
08   }
09 };
10 var roommates: Array<Person>[1..5];
11 rule step for person in roommates {
12   match person {
13     Happy {
14       person = Hungry;
15     }
16     Hungry {
17       if bananas == 0 {
18         person = GoingToStore;
19       } else {
20         bananas -= 1;
21         person = Happy;
22       }
23     }
24     GoingToStore {
25       person = ReturningFromStore {
26         carrying: urandomRange(0, 8)
27       };
28     }
29     ReturningFromStore(bag) {
30       bananas += bag.carrying;
31       person = Hungry;
32     }
33   }
34 }
35 invariant BananaLimit {
36   assert bananas <= 8;
37 }
```

For the purpose of this model, it's never OK to have more than eight bananas at home. This is checked by the invariant on lines 35–37. More sophisticated models could factor in a rate of decay and rate of consumption, but let's start simple.

Lines 1–10 declare two variables: "bananas" is the number of bananas at home, and "roommates" represents the five people who live there, each of whom is in one of various possible states at any given time. By default, Runway initializes variables to the upper-left possible value, so "bananas" starts at 0 and each person starts out "Happy."

Lines 11–34 declare a state transition rule named "step," which applies to one roommate at a time. If that person is "Happy," they can become "Hungry" (lines 13–15). If they are "Hungry" and a banana is available, they can eat a banana and become "Happy"; if no banana is available, they can go to the store (lines 16–23). If they're going to the store, they can return from the store with a random number of bananas between 0 and 8 (bunches vary in size, and sometimes the store has run out; lines 24–28). And if they are coming back from the store, they can leave the bananas they've purchased at home and return to being "Hungry," where they are likely to eat a banana soon (lines 29–32).

Note that specifications define which state transitions may happen, but they do not say when they should happen or in what order: that's up to the simulator or the model checker. If the specification permits multiple roommates to take a step from a given state, Runway's simulator will pick one at random. Alternatively, Runway's model checker will explore all possibilities, looking for any state that violates the invariant. To use the model checker with this specification, replace the random number on line 26 with a constant.

Inside Runway Views

Runway relies on a model's view to draw the main component of the visualization. Views are built using off-the-shelf Web technologies, so that Runway visualizations can run in a Web browser (the ubiquitous graphical toolkit). Although views are not necessarily constrained to these technologies, we're currently using JavaScript, SVG, and D3.js [4]:

- JavaScript is the scripting language running in every Web browser.
- SVG, Scalable Vector Graphics, is analogous to HTML but used for images instead of text and layout. Just like HTML, SVG is styled with CSS to assign properties like colors and borders.
- D3.js is a JavaScript library that assists with drawing SVG.

At a first approximation, specification and views tend to be similar in size. However, they are very different in nature. A view serves the necessary function of drawing the model's state, and its implementation tends to be uninteresting. You might study a specification to learn about the precise workings of a design, but the only thing you can learn from a view's code is how it draws the state.

Modeling Distributed Systems

Beyond simple banana problems, Runway can be used to model concurrent and distributed systems. Modeling distributed systems, in particular, introduces a new set of challenges. This section describes Runway's approach to modeling failures, networks, and clocks in distributed systems, as well as using invariants and assertions effectively to check properties of distributed system models.

Failures

In most distributed systems, servers can fail, and, down to some limit, the system should remain available. Messages can be delayed and perhaps dropped and reordered. These failures can be extremely important to understanding and evaluating designs, but different systems make different assumptions about their environments and have different requirements. In Runway, failures are encoded the same way as normal events, permitting specifications to model their own assumptions. A server crashing is modeled the same as a client submitting a request.

However, transition rules representing failures and client requests are different from normal transition rules in one regard: typically, they should not be applied all the time. For example, not every message should be dropped, and client requests should arrive at a limited rate. Currently, the specification can limit the rates of these events by imposing additional conditions on them, using random values. For example, when a message is sent, the specification can compute whether or when it will be dropped based on a coin toss. However, this need is recurring and fundamental to modeling, so we're exploring ways to express these event rates intuitively and conveniently in Runway.

Networks

Modeling a distributed system also requires modeling a network. This, too, can be done in Runway using normal state variables and transition rules. For example, the basic Raft model has a flat network modeled as a set. When a server sends a message, the message is added to the set. When a server receives a message, it is removed from the set.

Visualizing a network introduces its own challenge. For example, the Raft view draws each message as it moves from the sender to the recipient. To calculate the position of a message, it needs to know when the message will be received, but that information isn't normally available ahead of time. The Raft model currently takes a simple approach: the specification assigns each message a randomized delay when it is sent and will not deliver the message before then. An alternative, more complex approach would be to delay the visualization until the message's future delivery time had been determined by the simulator; we will implement this in Runway only if the simpler approach is found to be insufficient for common use cases.

In principle, more complex networks with links, switches/routers, and propagation and queueing delays can be modeled the same as simple networks, using variables and rules. However, as the network's wiring complexity increases, it would be tedious to express the wiring in Runway today, and we may explore additional language features to make this more convenient.

Runway also needs a way to import reusable components. This would be useful at various levels of scale, including:

- Choosing from several network models to load into a distributed system model,
- Loading a model of, for example, a coordination service into a model of a larger system, and
- Loading larger system models together into a model of an entire cluster's workload.

We are currently designing the language features to enable this.

Time and Clocks

Runway supports two modes of operation: *synchronous*, which generates events over time, and *asynchronous*, which generates only an ordered sequence of events. These two modes of operation have been useful for different models. For example, the basic Too Many Bananas model is asynchronous, while the model of a building's elevator system is primarily interesting to measure delays in synchronous mode.

The two modes can also be useful for the same model. Many algorithms are designed to maintain safety properties under asynchronous assumptions, making them robust to erroneous clocks and unexpected delays (typically, even a "small" race condition is not acceptable). With Runway, it's possible to check these properties asynchronously with the model checker and still run timing-based simulations on the same model. For example, we can check that Raft always keeps committed log entries no matter how long communication steps take, then use the same model in synchronous mode to estimate leader election times on a datacenter network.

In Runway, a timer is typically modeled by storing the time when an action should be taken, then guarding a transition rule for the action with an if-statement:

```
rule fireTimer {
  if past(timeoutAt) {
    /* take action, reset timeoutAt */
  }
}
```

This is another example of making all state explicit in Runway. There is no question of whether or not a timer has been set.

When running in synchronous mode, Runway keeps a global clock for the simulation, and “past()” evaluates to true if the given timestamp is earlier than the simulation’s clock. In asynchronous mode, however, “past()” always returns true. This has the effect of making all timers fireable immediately after they are scheduled, allowing unlikely schedules to be explored.

We have only tried Runway’s current approach to clocks on a handful of models, and it may need further enhancements. Specifically, it may be burdensome to model clock drift across servers using one global clock, as Runway provides today. We plan to revisit this issue based on actual use cases.

Access Restrictions and Distributed Invariants

Runway expects you to follow two ground rules in modeling distributed systems, but to keep the specification language simple, it does not enforce these rules. First, one server should not access another server’s internal state. Second, the only shared state should be the network, which should only be accessed in limited ways (such as following send/receive semantics). It would be impossible to implement a real distributed system that violated these rules.

However, accessing unshared state is OK for assertions and invariants. In fact, it’s a key advantage to modeling an entire distributed system in a single process. For example, in Raft there should be at most one leader per term. This is easy to check in an invariant by directly accessing and comparing all the servers’ states. The alternative, to check this property by exchanging messages as in a truly distributed system, would be much more complex, would be less effective due to message delays, and could interfere with the normal operation of the model.

Defining *history variables* as shared global state is also OK. History variables record information about the past. These variables should not affect the normal execution of the model, but they may be read by assertions and invariants. For example, Raft’s property that there is at most one leader per term should actually hold across time. If one server was leader in a particular term, no other server should ever become leader in that term. The Raft model tracks past leaders using a history variable, and when a server becomes leader in some term, it asserts that that term has not yet had a leader:

```
var electionsWon : Array<Boolean>[Term];
rule becomeLeader for server in servers {
  if (/* this candidate has a majority of votes */) {
    assert !electionsWon[server.term];
    electionsWon[server.term] = True;
    /* update local state to become leader */
  }
}
```

Conclusion

Distributed systems are challenging, and their complexity justifies careful design. Using the proper tools, we could be communicating clearly and evaluating our designs thoroughly, even before investing in their implementation. However, existing design tools have not been adopted widely in industry.

Runway hopes to change that. It combines specification, model checking, simulation, and interactive visualization into one tool. This improves Runway’s potential benefit without significantly increasing the cost of developing a model. Runway aims to be easy to learn by using a specification language based on imperative, procedural code that discourages unnecessary abstractions. Runway models are also easily shareable on the Web, so others can learn about designs through interactive visualization, even if they have not learned how to read Runway specifications.

At Salesforce, we are redesigning our infrastructure for the next order of scale, and we’ve already been applying Runway to a few design challenges internally. We found Runway to be effective for concurrent problems as well as distributed ones, and, encouragingly, engineers seem to find value early in specifying their designs more formally and in watching them run.

Runway is open source [2] and still in the early stages of its development. We have made it available early to find out whether other engineers will adopt it, and if not, to learn what is stopping them. We hope you will join us in forming a community around Runway.

Acknowledgments
Thanks to Rik Farrow, Pat Helland, Jennifer Wolochow, and Nat Wyatt (Runway's first user) for their helpful comments on earlier versions of this article, and to David Leon Gil, Steve Sandke, and many others for helping design Runway itself.

References
[1] Runway Wiki, Related Work: https://github.com/salesforce/runway-browser/wiki/Related-Work.

[2] Runway source code: https://github.com/salesforce/runway-browser.

[3] Raft Consensus Algorithm: https://raft.github.io.

[4] D3: Data Driven Documents, JavaScript library: https://d3js.org.

Design Guidelines for High Performance RDMA Systems

ANUJ KALIA, MICHAEL KAMINSKY, AND DAVID G. ANDERSEN

Anuj Kalia is a PhD student in the Computer Science Department at Carnegie Mellon University. He is interested in networked systems, especially using high-speed networks to build distributed systems. akalia@cs.cmu.edu

Michael Kaminsky is a Senior Research Scientist at Intel Labs and an adjunct faculty member of the Computer Science Department at Carnegie Mellon University. He is part of the Intel Science and Technology Center for Cloud Computing (ISTC-CC), based in Pittsburgh, PA. His research interests include distributed systems, operating systems, and networking. michael.e.kaminsky@intel.com

David G. Andersen is an Associate Professor of Computer Science at Carnegie Mellon University. He completed his SM and PhD degrees at MIT, and holds BS degrees in biology and computer science from the University of Utah. In 1995, he co-founded ArosNet, an ISP in Salt Lake City, Utah. dga@cs.cmu.edu

Modern RDMA hardware offers the potential for exceptional performance, but achieving this performance is challenging. Directly mapping an application's low-level reads and writes to RDMA primitives is often suboptimal, and design choices, including which RDMA operations to use and how to use them, significantly affect observed performance. We lay out guidelines that can be used by system designers to navigate the RDMA design space. Our guidelines emphasize paying attention to low-level details such as individual RDMA packets, PCIe transactions, and NIC architecture. We present two case studies—a key-value store and a networked sequencer—demonstrating the effectiveness of these guidelines.

In recent years, new entrants into the datacenter and cluster networking space have started to provide hardware capabilities formerly available only in expensive High Performance Computing (HPC) interconnects. The NICs (network interface cards) from manufacturers such as Mellanox now support RDMA (remote direct memory access) features out of the box, at a price comparable to non-RDMA-capable NICs.

The "RDMA Background" section describes RDMA in more detail, but at a high level RDMA is a networking approach consisting of two basic concepts:

1. Operating system "stack bypass": In many applications, the overhead of going through the kernel networking layers is the bottleneck to processing speed. This is particularly the case with applications that send and receive relatively small amounts of data per packet exchange, but do so at high rates.
2. Full CPU bypass: For certain, more-specialized applications, RDMA hardware can allow one computer to read and write directly to/from the memory of another node in the cluster, without the remote node's CPU or OS being involved at all.

RDMA has been a key ingredient of HPC and supercomputing environments for years, but it is also intriguing to datacenter application developers. RDMA hardware presents programmers with numerous choices, so using it efficiently requires care. For example, should applications provide reliability, or should the NIC's reliability protocol be used? In the rest of this article, we help readers navigate this space to understand what RDMA capabilities might be the best match for their application. Our open-source rdma_bench toolkit (https://github.com/efficient/rdma_bench) can be used for evaluating and optimizing the most important system factors that affect end-to-end throughput when using RDMA.

RDMA Background

RDMA is a general approach to networking for which several different models exist. The most popular model is the Virtual Interface Architecture (VIA) [3]. Other models include open-source specifications such as Portals from Sandia Labs, and proprietary HPC architectures such as Fujitsu's Tofu interconnect. For a given model, there can be more than one implementation. For example, VIA has three well-known implementations: InfiniBand, RoCE (RDMA over Converged Ethernet), and iWARP (Internet Wider Area RDMA Proto-

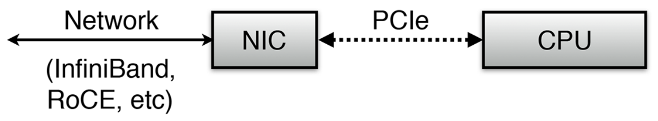


Figure 1: Hardware components of a node in an RDMA cluster

Verb	Abstract API function call
WRITE	write(qp, local_buf, size, remote_addr)
READ	read(qp, local_buf, size, remote_addr)
SEND	send(qp, local_buf, size)
RECV	recv (qp, local_buf, size)

Table 1: Abstract RDMA API showing one-sided (WRITE, READ) and two-sided (SEND, RECV) verbs

col). Our work focuses on VIA-based NICs, which are the only commodity NICs currently available. Several observations are applicable to other architectures as well. Table 1 shows an abstract RDMA API for VIA NICs.

Compared to conventional Ethernet-based TCP/IP networking, RDMA networks remove several sources of CPU overhead. They support user-level NIC access, removing the overhead of the kernel's heavyweight networking stack. At the remote host, RDMA reads and writes bypass the CPU. RDMA NICs typically implement a reliable transport layer, freeing up host CPU cycles used for implementing a reliable protocol such as TCP/IP. RDMA-capable networks with 100 Gbps of per-port bandwidth, and 2 µs round-trip latency are commercially available.

Figure 1 shows the relevant hardware components of a machine in a modern RDMA cluster. A NIC with one or more ports connects to the PCIe controller of a multicore CPU, and provides direct access to the memory of remote nodes.

Verb types: *One-sided* verbs (RDMA operations) operate directly on a remote node's memory, bypassing its CPU, and include RDMA reads, writes, and atomic operations. *Two-sided* verbs include the send and receive verbs; their functionality resembles `send()` and `recv()` functions in traditional sockets programming. These verbs involve the responder's CPU: the send's payload is written to an address specified by a receive that was posted previously by the responder's CPU.

The choice of verbs is a key determinant of application performance, but it is not the only factor. The choice of transport and the verb initiation method (discussed below) require equal attention.

Queue pairs: RDMA hosts communicate by posting verbs to interfaces called queue pairs (QPs). On completing a verb, the requester's NIC optionally signals completion by writing a completion queue entry to host memory via direct memory access (DMA).

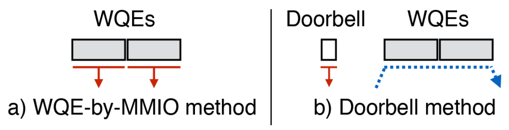


Figure 2: The WQE-by-MMIO and Doorbell methods for transferring two WQEs. Arrows represent PCIe transactions. Solid arrows are PCIe MMIO writes; the dashed arrow is a PCIe DMA read. Arrow width represents transaction size.

RDMA transports are either reliable or unreliable and are either connected or unconnected (also called datagram). With reliable transports, the NIC uses acknowledgments to guarantee in-order delivery of messages. Unreliable transports do not provide this guarantee. However, modern high-speed networks such as InfiniBand and RoCE use a reliable link layer, so unreliable transports do not lose packets due to congestion or bit errors. Connected transports require one-to-one connections between QPs, whereas a datagram QP can communicate with multiple QPs. Datagram transport is more scalable, but it only supports two-sided verbs.

Current RDMA transports include Reliable Connected (RC), Unreliable Connected (UC), and Unreliable Datagram (UD). Note that although these transports resemble non-RDMA transport layers to some extent (e.g., RC and UD are analogous to TCP and UDP, respectively), the underlying protocol and message formats are different.

Verb initiation: To initiate RDMA operations, the user-mode NIC driver at the requester creates work queue elements (WQEs) in host memory. These WQEs are transferred to the NIC over the PCIe bus in one of two ways. In the "WQE-by-MMIO" method, the CPU directly writes the WQEs to device memory using memory-mapped I/O (MMIO). In the "Doorbell" method, the CPU writes a short Doorbell message to the NIC, indicating the new WQEs. This action is called "ringing the Doorbell." On receiving the Doorbell, the NIC reads the WQEs from the CPU via a DMA read. Both methods bypass the host's OS kernel. Figure 2 summarizes the two methods. The Doorbell method reduces CPU use: it requires one MMIO for a batch of WQEs, whereas WQE-by-MMIO requires separate MMIOs for each WQE.

Factors Affecting RDMA System Performance

In datacenters, RDMA is being proposed for use in key-value stores, graph processing systems, and online transaction processing systems [1, 2]. These applications access irregular data structures (e.g., hash tables and trees) and use small packet sizes on the order of tens of bytes. Three main factors are important for high performance with these workloads: the extent to which remote CPU bypass is used, low-level optimizations for verbs, and the NIC architecture.

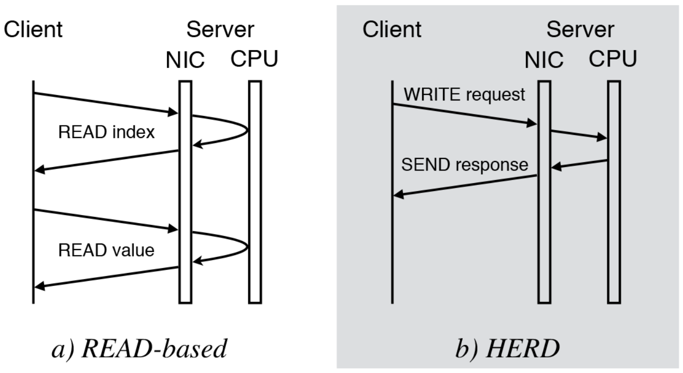


Figure 3: Messages for a GET operation in (a) a READ-based key-value store and (b) HERD

Remote CPU Bypass

In an RDMA-based datastore, data is stored in the memory of a server machine and is accessed by client machines. To take advantage of RDMA's ability to bypass the remote CPU, several projects use one-sided READs and WRITEs to accomplish this. For datastore GET operations, they do so by traversing the remote data structure using READs. This typically requires multiple round trips. For example, Pilaf [6] is an RDMA-based key-value store that handles key-value GET operations in 2.6 READs on average. It uses 1.6 READs to access its hash table–based index to locate the value's address, and one READ to fetch the value. FaRM's key-value store [1] reduces the number of READs required for the index from 1.6 to 1.

In contrast, the design of our HERD system [4] is focused on reducing the number of round trips to one. To accomplish this, HERD does not entirely bypass the remote CPU. Instead of traversing the remote data structure, HERD clients send their requests to the server using an RPC request. The HERD server traverses the data structure for the client, but it does so in local memory.

Local memory accesses are 10x–100x faster than remote accesses in latency, bandwidth, and the amount of host CPU they consume. Then the server sends a response to the client. Several combinations of verbs and transports can be used to implement fast RPCs; HERD uses a combination of one-sided and two-sided verbs over unreliable transport, which is optimized for high performance and number-of-clients scalability. Figure 3 shows the difference in HERD and READ-based key-value stores.

HERD's RPC mechanism is very fast: its throughput and latency is similar to RDMA reads. As a result, HERD delivers higher throughput and lower latency than Pilaf or FaRM's key-value store. Key to achieving high RPC throughput is the use of the low-level optimizations discussed below. An important lesson from HERD is that one-sided RDMA is not always the best solution; using an RDMA network simply for fast, OS kernel–bypass RPCs is an equally important design to consider.

Low-Level Verb Optimizations

RDMA verbs allow for a variety of low-level optimizations. Effectively using these optimizations requires a good understanding of how different verbs use the CPU, the PCIe bus, and the RDMA network, and how this varies when different optimizations are enabled. Our USENIX ATC paper [5] addresses this topic thoroughly. We discuss the most important optimizations briefly here.

Unreliable transports reduce NIC overhead by not requiring RDMA acknowledgment packets, and provide higher performance than reliable transports. Unreliable transports do not provide reliable packet delivery. However, modern RDMA implementations such as InfiniBand use a reliable link layer, so even unreliable transports drop packets extremely rarely.

Payload inlining reduces NIC processing and PCIe bandwidth use by eliminating the DMA read for the payload. By default, WRITEs and SEND work queue elements contain a pointer to the payload; the NIC reads it via a DMA read. Small payloads up to a few hundred bytes can be encapsulated inside the WQE and written to the NIC using MMIO.

Selective signaling also reduces NIC processing and PCIe bandwidth use by eliminating the completion DMA. By default, the NIC writes a completion queue entry to host memory on completing a verb. If an application can detect completion using alternate methods (e.g., using a future response from a remote node), it can mark the verb as "unsignaled," instructing the NIC to not issue the completion DMA.

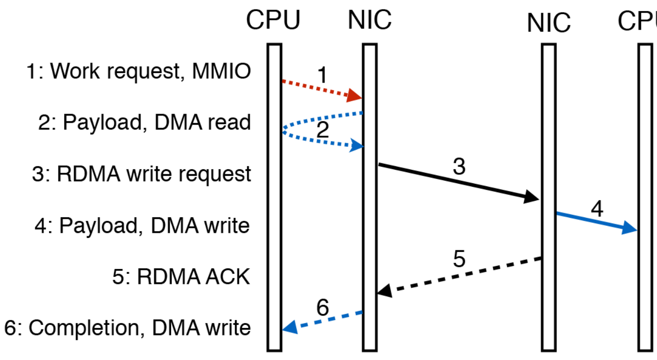
Doorbell batching reduces CPU use and PCIe bandwidth use by allowing applications to issue a batch of RDMA operations using one Doorbell. This reduces CPU use by requiring only one MMIO per batch. The default approach of transferring WQEs one by one using WQE-by-MMIO requires separate MMIOs for each WQE.

Figure 4a and Figure 4b show the PCIe and RDMA network messages for one WRITE without optimizations, and two WRITEs with the above optimizations, respectively.

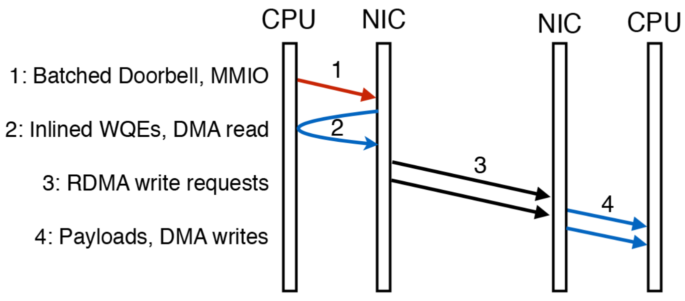
NIC Architecture

Modern high-speed NICs are composed of multiple processing units (PUs), such as packet processing engines and DMA engines. Exploiting parallelism among the NIC's PUs is necessary for high performance but requires explicit attention. Further, RDMA verbs and workloads that introduce contention between the PUs should be avoided.

Engage multiple NIC PUs: A common RDMA programming decision is to use as few queue pairs as possible, but doing so limits NIC parallelism to the number of QPs. This is because operations on the same QP have ordering dependencies and are ideally handled by the same NIC processing unit to avoid cross-



(a) RDMA and PCIe messages for one RDMA write



(b) RDMA and PCIe messages for two WRITEs with optimizations

Figure 4: Effect of optimizations on RDMA and PCIe messages. The optimized WRITEs are unreliable, inlined, unsignaled, and are issued using a batched Doorbell. The dashed messages are removed in the optimized version; the dotted messages are repurposed.

PU synchronization. For example, in datagram-based RDMA communication, one QP per CPU core is sufficient for communication with all remote cores. However, it "binds" a CPU core to a PU and may limit core throughput to PU throughput. This is likely to happen when per-message application processing is small and a high-speed CPU core overwhelms a less powerful PU. In such cases, using multiple QPs per core increases CPU efficiency; we call this the *multi-queue* optimization.

Avoid contention among NIC PUs: RDMA operations that require cross-QP synchronization introduce contention among PUs, and can perform over an order of magnitude worse than uncontended operations. For example, RDMA provides atomic operations such as compare-and-swap and fetch-and-add on remote memory. To our knowledge, all commodity NICs available at the time of writing use internal concurrency control for atomics: PUs acquire an internal lock for the target address and issue read-modify-write over PCIe. Therefore, the NIC's internal locking mechanism, such as the number of locks and the mapping of atomic addresses to these locks, is important. Note that due to the limited SRAM in NICs, the number of available locks is small, which amplifies contention in the workload.

Case Studies

We now describe the design of two high-performance RDMA-based systems: the HERD key-value store and the X-Seq sequencer. X-Seq is named Spec-S0 in our USENIX ATC paper [5].

RPC Overview

For both systems, we use an RPC protocol for communication between clients and the key-value/sequencer server. In HERD, clients use unreliable WRITEs to write requests to a request memory region at the server. A server thread (a *worker*) checks for new requests from every client by polling on the request memory region, and collects a batch of requests. It computes a batch of responses, and sends them to clients using the SEND verb over datagram transport. The worker uses a batched Doorbell for the batch of response SENDs. To use the multi-queue optimization, each worker alternates among a configurable number (1–3) of datagram QPs across batches of response SENDs. In addition to unreliable transport, Doorbell batching, and multi-queue, the server also uses payload inlining and selective signaling.

Key-Value Stores

Figure 5 shows the throughput of a HERD key-value store server with an increasing number of server CPU cores. We use a cluster with Mellanox Connect-IB InfiniBand NICs, and 14-core Intel CPUs. The key-value store maps 16-byte keys to 32-byte values; the workload consists of keys chosen uniformly at random and 5% PUT operations. HERD's single-core throughput is 12.3 million operations/s (Mops), and its peak throughput is 98.3 Mops. At its peak, HERD is bottlenecked by PCIe bandwidth.

Figure 5 also compares HERD to a READ-based key-value store that requires two RDMA reads per GET operation. The Connect-IB NIC supports up to 120 million READs/s, so such a READ-based key-value store is limited to 60 Mops. HERD delivers up to 64%

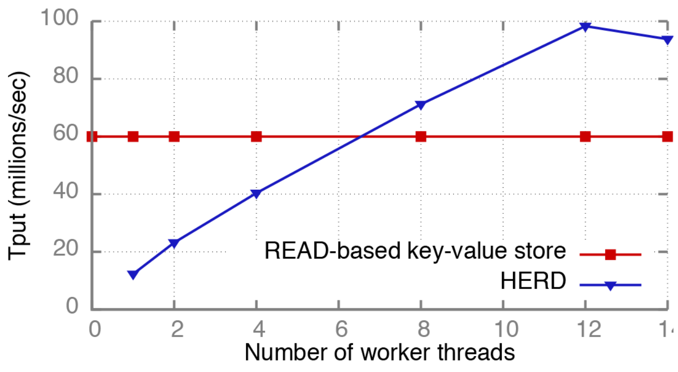


Figure 5: Throughput of HERD and READ-based key-value stores

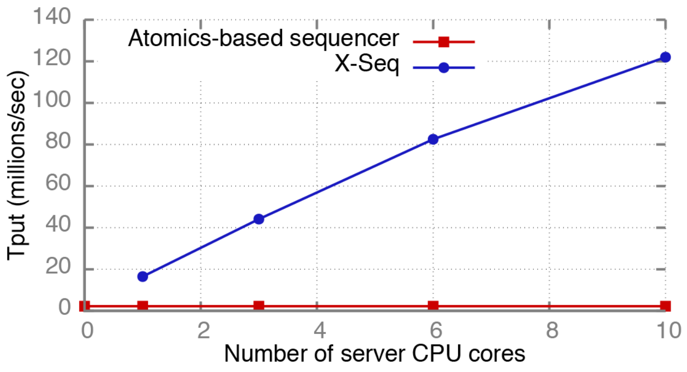


Figure 6: Throughput of X-Seq and atomics-based sequencers

higher throughput, while using a single round trip per operation. HERD uses significant server CPU resources: it requires at least seven CPU cores to outperform a READ-based design. However, HERD uses less client CPU than a READ-based store because it requires fewer client-initiated data transmissions.

Networked Sequencers

Centralized sequencers are useful building blocks for a variety of network applications, such as ordering operations in distributed systems via logical or real timestamps. A centralized sequencer can be the bottleneck in high-performance distributed systems, so building a fast sequencer is an important step to improving whole-system performance.

Our X-Seq sequence server runs on a single machine and provides an increasing eight-byte integer to client processes running on remote machines. The worker threads at the server share an eight-byte counter. After collecting a batch of *N* client requests, a worker thread atomically increments the shared counter by *N*, thereby claiming ownership of a sequence of *N* consecutive integers. It then sends these *N* integers to the clients using a batched Doorbell (one integer per client).

Directly adapting HERD's RPC protocol to a sequencer provides good performance. However, even higher throughput and scalability can be achieved by optimizing the RPCs specifically for the sequencer. The key insight is that the request and response packets in the sequencer are small, with up to eight bytes of data. This allows RPC optimizations that reduce the number of cache lines used by WQEs, and DMAs issued, by 50%. We also use a speculation technique where the clients speculate the most significant bytes of the current sequencer number.

Figure 6 shows the throughput of X-Seq with increasing server CPU cores. Its single-core throughput is 16.5 Mops, and its peak throughput is 122 Mops. Figure 6 also shows the throughput of a sequencer where clients use the atomic fetch-and-add verb to increment an eight-byte counter in the server's memory. This sequencer achieves only 2.24 Mops.

The poor performance of the atomics-based sequencer is because of lock contention among the NIC's processing units. The effects of contention are exacerbated by the *duration* for which locks are held—several hundred nanoseconds for PCIe round trips. Our RPC-based sequencers have lower contention and shorter lock duration: the programmability of general-purpose CPUs allows us to batch updates to the counter, which reduces cache line contention, and proximity to the counter's storage (i.e., core caches) makes these updates fast.

Code release: The code for our low-level RDMA benchmarks, HERD, and X-Seq is available at https://github.com/efficient/rdma_bench.

References

[1] A. Dragojević, D. Narayanan, O. Hodson, and M. Castro, "FaRM: Fast Remote Memory," in *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI '14),* 2014.

[2] A. Dragojević, D. Narayanan, E. B. Nightingale, M. Renzelmann, A. Shamis, A. Badam, and M. Castro, "No Compromises: Distributed Transactions with Consistency, Availability, and Performance," in *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP '15),* 2015.

[3] D. Dunning, G. Regnier, G. McAlpine, D. Cameron, B. Shubert, F. Berry, A. M. Merritt, E. Gronke, and C. Dodd, "The Virtual Interface Architecture," *IEEE Micro,* 1998, pp. 66–76.

[4] A. Kalia, M. Kaminsky, and D. G. Andersen, "Using RDMA Efficiently for Key-Value Services," in *Proceedings of ACM SIGCOMM,* 2014, pp. 295–306.

[5] A. Kalia, M. Kaminsky, and D. G. Andersen, "Design Guidelines for High-Performance RDMA Systems," in *Proceedings of the 2016 USENIX Annual Technical Conference (ATC '16).*

[6] C. Mitchell, Y. Geng, and J. Li, "Using One-Sided RDMA Reads to Build a Fast, CPU-Efficient Key-Value Store," in *Proceedings of the 2013 USENIX Annual Technical Conference (ATC '13).*

ENIGMA

MORE TO DECIPHER

It's time for the security community to take a step back and get a fresh perspective on threat assessment and attacks. This is why in 2016 the USENIX Association launched Enigma, a new security conference geared towards those working in both industry and research. Enigma will return in 2017 to keep pushing the community forward.

Expect three full days of high-quality speakers, content, and engagement for which USENIX events are known.

The full program and registration will be available in October.

enigma.usenix.org

JAN 30–FEB 1 2017

OAKLAND, CALIFORNIA, USA

usenix ASSOCIATION

SYSADMIN

Invent More, Toil Less

BETSY BEYER, BRENDAN GLEASON, DAVE O'CONNOR, AND VIVEK RAU

Betsy Beyer is a Technical Writer for Google Site Reliability Engineering in NYC. She has previously written documentation for Google Datacenters and Hardware Operations teams. Before moving to New York, Betsy was a lecturer on technical writing at Stanford University. She holds degrees from Stanford and Tulane.
bbeyer@google.com

Brendan Gleason is a Site Reliability Engineer in Google's NYC office. He has worked on several Google storage systems and is currently bringing Bigtable to the cloud. Brendan has a BA from Columbia University. bfg@google.com

Dave O'Connor is an SRE Manager at Google Dublin, responsible for Google's shared storage and the Production Network. He previously worked for Netscape and AOL in Ireland, as well as for several smallish startups in Dublin. He holds a BSc in Computer Applications from Dublin City University. daveoc@google.com

Vivek Rau is an SRE Manager at Google and a founding member of the Launch Coordination Engineering sub-team of SRE. His current focus is improving the reliability of Google's cloud platform. Vivek has a BS degree in computer science from IIT-Madras. vivekr@google.com

This article builds upon Vivek Rau's chapter "Eliminating Toil" in *Site Reliability Engineering: How Google Runs Production Systems* [1]. We begin by recapping Vivek's definition of toil and Google's approach to balancing operational work with engineering project work. The Bigtable SRE case study then presents a concrete example of how one team at Google went about reducing toil. Finally, we leave readers with a series of best practices that should be helpful in reducing toil no matter the size or makeup of the organization.

SRE's Approach to Toil

As discussed in depth in the recently published *Site Reliability Engineering*, Google SRE seeks to cap the time engineers spend on operational work at 50%. Because the term *operational work* might be interpreted in a variety of ways, we use a specific word to describe the type of work we seek to minimize: *toil*.

Toil Defined

To define toil, let's start by enumerating what toil is *not*. Toil is not simply equivalent to:

- "Work I don't like to do"
- Administrative overhead such as team meetings, setting and grading goals, and HR paperwork
- Grungy work, such as cleaning up the entire alerting configuration for your service and to remove clutter

Instead, toil is the kind of work tied to running a production service that tends to be:

- Manual
- Repetitive
- Automatable and not requiring human judgment
- Interrupt-driven and reactive
- Of no enduring value

Work with enduring value leaves a service permanently better, whereas toil is "running fast to stay in the same place." Toil scales linearly with a service's size, traffic volume, or user base. Therefore, as a service grows, unchecked toil can quickly spiral to fill 100% of everyone's time.

As reported by SREs at Google, our top three sources of toil (in descending order) are:

- Interrupts (non-urgent service-related messages and emails)
- On-call (urgent) responses
- Releases and pushes

Toil isn't always and invariably bad; all SREs (and other types of engineers, for that matter) necessarily have to deal with some amount of toil. But toil becomes toxic when experienced in large quantities. Among the many reasons why too much toil is bad, it tends to lead to career stagnation and low morale. Spending too much time on toil at the expense of time spent engineering hurts the SRE organization by undermining our engineering-focused mission, slowing progress and feature velocity, setting bad precedents, promoting attrition, and causing breach of faith with new hires who were promised interesting engineering work.

Addressing Toil through Engineering

Project work undertaken by SREs is key in keeping toil at manageable levels. Capping operational work at 50% frees up the rest of SRE time for long-term engineering project work that aims to either reduce toil or add service features. These new features typically focus on improving reliability, performance, or utilization—efforts which often reduce toil as a second-order effect.

SRE engineering work tends to fall into two categories:

- **Software engineering:** Involves writing or modifying code, in addition to any associated design and documentation work. Examples include writing automation scripts, creating tools or frameworks, adding service features for scalability and reliability, or modifying infrastructure code to make it more robust.
- **Systems engineering:** Involves configuring production systems, modifying configurations, or documenting systems in a way that produces lasting improvements from a one-time effort. Examples include monitoring setup and updates, load-balancing configuration, server configuration, tuning of OS parameters, and load-balancer setup. Systems engineering also includes consulting on architecture, design, and productionization for developer teams.

Engineering work enables the SRE organization to scale up sublinearly with service size and to manage services more efficiently than either a pure Dev team or a pure Ops team.

Case Study: Bigtable SRE

It's important to understand exactly *what* toil is, and why it should be minimized, before engaging boots on the ground to address it. Here's how one SRE group at Google actively worked to reduce toil once they realized that it was overburdening the team.

Toil in 2012

In 2012, the SRE team responsible for operating Bigtable, a Google high performance data storage system, and Colossus, the distributed file system upon which Bigtable was built, was suffering from a high rate of operational load.

Early in the year, pages had reached an unsustainable level (five incidents per standard 12-hour shift; Google purposefully designs many of its SRE teams to be split across two sites/time zones to provide optimal coverage without overtaxing on-call engineers with 24-hour shifts), and the team began an effort to eliminate unnecessary alerts and address true root causes of pages. With concentrated effort, the team brought the pager load down to a more sustainable level (around two incidents per shift). However, incident response was only one component of the team's true operational load. User requests for quota changes, configuration changes, performance debugging, and other operational tasks were accumulating at an ever-increasing rate. What began as a sustainable support model when Bigtable SRE was responsible for just a few cells and a handful of customers had snowballed into an unpleasant amount of unrewarding toil.

The team wasn't performing all of its daily operations "by hand," as SREs had created partial automation to assist with a number of tasks. However, this automation stagnated while both the size of Google's fleet and the number of services that depend on Bigtable grew significantly. On any given day, multiple engineers were involved in handling the toil-driven work that resulted from on-call incidents and customer requests, which meant that these SREs couldn't focus on engineering and project work. In fact, an entire subteam was dedicated to the repetitive but obligatory task of handling requests for increases and decreases in Bigtable capacity. To make matters worse, the team was so overburdened with operational load that they didn't have time to adequately root cause many of the incidents that triggered pages. The inability to resolve these foundational problems created a vicious cycle of ever-increasing operational load.

Turning Point

Acknowledging that its operational trajectory was unsustainable, the entire Bigtable and Colossus SRE team assembled to discuss its roadmap and future. While team members were nearly universally unhappy with the level of operational load, they also felt a strong responsibility to both support their users and to make Google's storage system easy to use. They needed a solution that would benefit all parties involved in the long run, even if this solution meant making some difficult decisions about how to proceed in the short term.

After much discussion, Bigtable SRE agreed that continuing to sacrifice themselves to achieve the short-term goals of their customers was actually counterproductive, not only the team, but also to their customers. While fulfilling customer requests on an as-needed basis might have been temporarily gratifying, it was not a sustainable strategy. In the long run, customers value a reliable, predictable interface offered by a healthy team more than they value a request queue that processes any and every request, be it standard or an unconventional one-off, in an indeterminate amount of time.

Tactics

The team realized that in order to get their operational work under control and improve the Bigtable service for their users, they would have to say "no" to some portion of customer requests for a period of time. The team, supported by management, decided that it was important (and ultimately better for Bigtable users) to respect their colleagues and themselves by pushing back on complex customer requests, performance investigations for customers who were within Bigtable's promised SLO, and other routine work that yielded nominal value. The team's management understood that the long-term health of both the team and the service could be substantially improved by making carefully considered short-term sacrifices in service quality.

Additionally, they decided to split the team into two shards: one focused on Bigtable, and one focused on Colossus. This split had two advantages: it allowed engineers to specialize technically on a single product, and it allowed the leads of each shard to focus on improving the operational state of a single service.

In addition to temporarily impacting how, and how quickly, they processed user requests, the team recognized that their new focus on reducing operational load would also impact their work in a couple of other key areas: their ability to complete project work and their relationship with partner developer teams. For the time being, SREs would have less bandwidth to collaborate with the core Bigtable development team in designing, qualifying, and deploying new features. Fortunately, the Bigtable developers anticipated that reducing operational load would result in a better, more stable product, and went so far as to allocate some of their engineers to this effort. Assisting the SRE team in improving service automation would ultimately benefit both teams if developers could shorten the window of slowed feature velocity.

The Turnaround Begins: Incremental Progress

Equipped with a narrowed scope and a clear mandate to focus on reducing toil, the Bigtable SRE Team began making progress in clearing their operational backlog. They first turned an eye to routine user requests. The overwhelming majority of requests fell into three buckets:

- Increases and decreases in quota
- Turnups and turndowns of Bigtable footprints
- Turnups and turndowns of datacenters

Rather than trying to engineer an all-encompassing big-bang solution, the team made an important decision: to deliver incremental progress.

Bigtable SRE first focused on fully automating the various footprint- and quota-related requests. While this step didn't eliminate tickets, it greatly simplified the ticket queue and

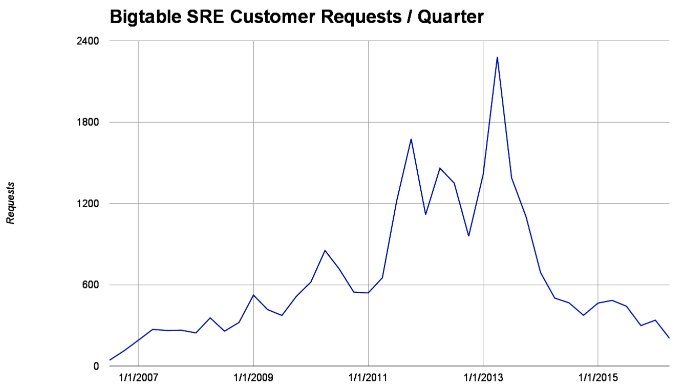


Figure 1: Bigtable SRE customer requests per quarter

reduced the amount of time it took to complete requests. The team could now fulfill each request by simply starting automation to complete the task, eliminating the several manual steps previously necessary.

Next, the team focused on wrapping automation into self-service tools. Initially, they simply added quota to an existing footprint, which was both the most common request and the easiest request to transition to self-service. SREs then began adding self-service coverage for more operations, prioritizing according to complexity and frequency. They tackled common and less complex tasks first, moving from quota reductions, to footprint turnups, to footprint turndowns.

Bigtable SRE's iterative approach was twofold: in addition to tackling lower-hanging fruit first, they approached each self-service task starting from the basics. Rather than trying to create fully robust solutions from the get-go, they launched basic functionality, upon which they incrementally improved. For example, the initial version of the self-service software for quota reductions and footprint turndowns couldn't handle all possible configurations. Once users were equipped with this basic functionality, the engineers incrementally expanded the self-service coverage to a growing fraction of the request catalog.

End Game

By breaking up the toil problem into smaller surmountable pieces that could deliver incremental value, Bigtable SRE was able to create a snowball of work reduction: each incremental reduction of toil created more engineering time to work on future toil reduction. As shown in Figure 1, by 2014, the team was in a much improved place operationally—they reduced user requests from a peak of more than 2200 requests per quarter in early 2013 to fewer than 400 requests per quarter.

Looking Forward

While Bigtable SRE significantly improved its handle on toil, the war against toil is never over. As Bigtable continues to add new features, and its number of customers and datacenters continues to grow, Bigtable SRE is constantly on the offensive in combatting creeping levels of toil. Perhaps the most significant change Bigtable SRE underwent in this process was a shift in culture. Before the turnaround, the team viewed operational work as an unpleasant but necessary task that they didn't have the power to refuse or delay. Since the turnaround, the team is extremely skeptical of any feature or process that will add operational work. As team members challenge and hold each other accountable for the level of operational load on the team, they aim to never regress to similarly undesirable levels of toil.

Best Practices for Reducing Toil

Now that we've seen how one particular SRE team at Google tackled toil, what lessons and best practices can you glean from a massive-scale operation like Google that apply to your own company or organization?

As they're tasked with running the entire gamut of services that make up Google production, SRE teams at Google are necessarily varied, as are their approaches to toil reduction. While some of the particular approaches taken by a team like Bigtable SRE might not be relevant across the board, we've boiled down SRE's diverse approaches to reducing toil into some essential best practices. These recommendations hold regardless of whether you're approaching service management from scratch or looking to help a team already burdened by excessive toil.

Buy-in Is Key

As demonstrated by the Bigtable SRE case study, you can't tackle toil in a meaningful way without managerial support behind the idea that toil reduction is a worthwhile goal. Sometimes long-term wins come with the tradeoff of short-term compromises, and securing managerial buy-in for temporarily pushing back on routine but important work is likely easier said than done. The key here is for management to consider what measures will enable a team to be significantly more effective in the long run. For example, Bigtable SRE was only able to rein in the toil overwhelming their team by deprioritizing feature development and manual and time-consuming customer requests in the short term.

Bigtable SRE also found that breaking down toil reduction efforts into a series of small projects was key for a few reasons. Perhaps most obviously, this incremental approach gives the team a sense of momentum early on as it meets goals. It also enables managers to evaluate a project's direction and provide course corrections. Finally, it makes progress easily visible, increasing buy-in from external stakeholders and leadership.

Minimize Unique Requirements

Using the "pets vs. cattle" analogy discussed in a 2013 UK *Register* article [2], your systems should be automated, easily interchangeable, replaceable, and low-maintenance (cattle); they should not have unique requirements for human care and attention (pets). Should disaster strike, you'll be in a much better position if you've created systems that can be recreated easily from scratch. Tempting as it might be to manually cater to individual users or customers, such a model is not scalable.

Similarly, understand the difference between parts of the system that require individual care and attention from a human versus parts that are unremarkable and just need to self-heal or be replaced automatically. Depending on your scale, these components might be hosts, racks of hosts, network links, or even entire clusters.

Be thoughtful about how you handle configuration management. By using a centrally controlled tool like Puppet, you gain scalability, consistency, reliability, reproducibility, and change management control over your entire system, allowing you to spin up new instances on demand or push changes en masse.

While many people and teams recognize that building one-off solutions is suboptimal, it's still often tempting to build such systems. Actually steering away from creating special cases for short-term efficacy and insisting on standardized, homogeneous solutions requires focus and periodic review by team leads and managers.

Invest in Build/Test/Release Infrastructure Early

Instituting standardization and automation might be a hard sell early on in a service's life cycle, but it will pay off many times over down the road. Implementing this infrastructure is much harder later on, both technically and organizationally.

That said, there's a balance between insisting on this approach wholesale, thus hurting velocity, versus postponing infrastructure development until suboptimally late in the development cycle. Try to plan accordingly—once you're beyond the rapid launch-and-iterate phase and relatively certain that the system will have the longevity to warrant this kind of investment, put sufficient time and effort into developing build, test, and release infrastructure.

Audit Your Monitoring Regularly

Establish a regular feedback loop to evaluate signal versus noise in your monitoring setup. Be thorough and ruthless in eliminating noisy and non-actionable alerts. Otherwise, important alerts that you *should* be paying attention to are drowned out in the noise. For each real-time alert, repeat the mantra, "What does a **human being** need to do, **right this second**?" The *Site Reliability Engineering* chapter "Monitoring Distributed Systems" covers this topic in depth.

Conduct Postmortems

The need for postmortems may not surface in the course of everyday work, but consistently undertaking them massively contributes to the stability of a system or service. Instead of just scrambling to get the system back up and running every time an incident occurs, take the time to identify and triage the root cause after the immediate crisis is resolved. As detailed in the *SRE* chapter "Postmortem Culture: Learning from Failure," these collaborative postmortem documents should be both blameless and actionable. Avoid one-size-fits-all approaches: this exercise should be lightweight for small and simple incidents but much more in-depth for large and complex outages.

No Haunted Graveyards

Even when it comes to companies and teams that consider themselves fast-moving and open to risk, parts of production or the codebase are sometimes considered "too risky" to change—either very few people understand these components or they were designed in such a way that there's a risk assigned to changing or touching them. Our goal is to control trouble, not to avoid it at all costs. In such cases of perceived risk, smoke out risk rather than leaving it to fester.

Conclusion

Any team tasked with operational work will necessarily be burdened with some degree of toil. While toil can never be completely eliminated, it can and should be thoughtfully mitigated in order to ensure the long-term health of the team responsible for this work. When operational work is left unchecked, it naturally grows over time to consume 100% of a team's resources. Engineers and teams performing an SRE or DevOps role owe it to themselves to focus relentlessly on reducing toil—not as a luxury, but as a necessity for survival.

The type of engineering work generated by toil reduction projects is much more interesting and fulfilling than operational work, and it leads to career growth and healthier team dynamics. Google SRE teams have found that working from the set of best practices above, in addition to constantly reassessing our workload and strategies, has equipped us to continually scale up the creative challenges, business impact, and technical sophistication of the SRE job.

Acknowledgments

Special thanks to the following for helping with background for this article: Olivier Ansaldi, Brent Chapman, Neil Crellin, Sandy Jensen, Jeremy Katz, Thomas Labatte, Lisa Lund, and Nir Tarcic.

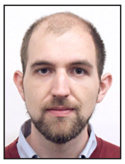
References

[1] B. Beyer, C. Jones, J. Petoff, and N. Murphy, eds., *Site Reliability Engineering* (O'Reilly Media, 2016).

[2] S. Sharwood, "Are Your Servers Cattle or Pets?" *The Register,* March 18, 2013: http://www.theregister.co.uk/2013/03/18/servers_pets_or_cattle_cern/.

Some Routes Are More Default than Others

JONATHON ANDERSON



Jonathon Anderson has been an HPC Sysadmin since 2006 and believes that everything would be a lot easier if we just spent more time figuring out the correct way to do things. He's currently serving as HPC Engineer at the University of Colorado, and hopes to stick around Boulder for a long time to come.
jonathon.anderson@colorado.edu

Typical IP-networked hosts are configured with a single default route. For single-homed hosts the default route defines the first destination for packets addressed outside of the local subnet; but for multi-homed hosts the default route also implicitly defines a default interface to be used for all outbound traffic. Specific subnets may be accessed using non-default interfaces by defining static routes; but the single default route remains a "single point of failure" for general access to other and Internet subnets. The Linux kernel, together with the iproute2 suite [1], supports the definition of multiple default routes distinguished by a preference metric. This allows alternate networks to serve as failover for the preferred default route in cases where the link has failed or is otherwise unavailable.

Background

The CU-Boulder Research Computing (RC) environment spans three datacenters, each with its own set of special-purpose networks. Public-facing hosts may be accessed through a 1:1 NAT or via a dedicated "DMZ" VLAN that spans all three environments. We have historically configured whichever interface was used for inbound connection from the Internet as the default route in order to support responses to connections from Internet clients; but our recent and ongoing deployment of policy routing (as described in the summer 2016 issue of *;login:*) removes this requirement.

All RC networks are capable of routing traffic with each other, the campus intranet, and the greater Internet, so we more recently prefer the host's "management" interface as its default route as a matter of convention; but this unnecessarily limits network connectivity in cases where the default interface is down, whether by link failure or during a reconfiguration or maintenance process.

The Problem with a Single Default Route

The simplest Linux host routing table is a system with a single network interface.

```
# ip route list
default via 10.225.160.1 dev ens192
10.225.160.0/24 dev ens192  proto kernel  scope link  src 10.225.160.38
```

Traffic to hosts on 10.225.160.0/24 is delivered directly, while traffic to any other network is forwarded to 10.225.160.1. In this case, the default route eventually provides access to the public Internet.

```
# ping -c1 example.com
PING example.com (93.184.216.34) 56(84) bytes of data.
64 bytes from 93.184.216.34: icmp_seq=1 ttl=54 time=24.0 ms

--- example.com ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 24.075/24.075/24.075/0.000 ms
```

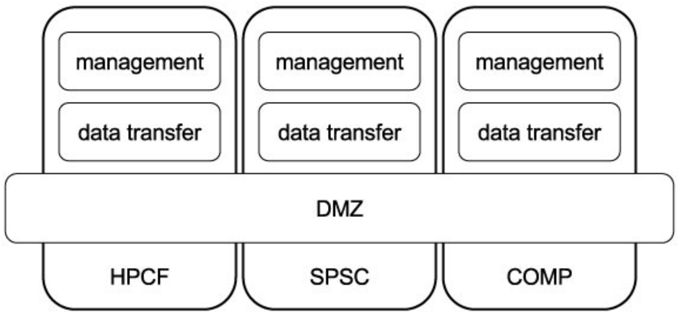


Figure 1: The CU-Boulder Research Computing Science Network, with subnets in three datacenters

A dual-homed host adds a second network interface and a second link-local route, but the original default route remains.

```
# ifup ens224 && ip route list
default via 10.225.160.1 dev ens192
10.225.160.0/24 dev ens192  proto kernel  scope link  src
10.225.160.38
10.225.176.0/24 dev ens224  proto kernel  scope link  src
10.225.176.38
```

The new link-local route provides access to hosts on 10.225.176.0/24, but traffic to other networks still requires access to the default interface as defined by the single default route. If the default route interface is unavailable, external networks become inaccessible, even though identical routing is available via 10.225.176.1.

```
# ifdown ens192 && ping -c1 example.com; ifup ens192
connect: Network is unreachable
```

Attempts to add a second default route fail with an error message (in typically unhelpful iproute2 fashion), implying that it is impossible to configure a host with multiple default routes simultaneously.

```
# ip route add default via 10.225.176.1 dev ens224
RTNETLINK answers: File exists
```

It would be better if the host could select dynamically from any of the physically available routes, but without an entry in the host's routing table directing packets out the `ens224` "data" interface, the host will simply refuse to deliver the packets.

Multiple Default Routes and Routing Metrics

The `RTNETLINK` error above indicates that the `ens224` "data" route cannot be added to the table because a conflicting route already exists—in this case, the `ens192` "management" route. Both routes target the "default" network, which would lead to non-deterministic routing with no way to select one route in favor of the other.

However, the Linux routing table supports more attributes than the "via" address and "dev" specified in the above example. Of use here, the "metric" attribute allows us to specify a preference number for each route.

```
# ip route change default via 10.225.160.1 dev ens192 metric 100
# ip route add default via 10.225.176.1 dev ens224 metric 200
# ip route flush cache
```

The host will continue to prefer the `ens192` "management" interface for its default route due to its lower metric number, but if that interface is taken down, outbound packets will automatically be routed via the `ens224` "data" interface.

```
# ifdown ens192 && ping -c1 example.com; ifup ens192
PING example.com (93.184.216.34) 56(84) bytes of data.
64 bytes from example.com (93.184.216.34): icmp_seq=1 ttl=54
time=29.0 ms

--- example.com ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 29.032/29.032/29.032/0.000 ms
```

Persisting the Configuration

This custom-routing configuration can be persisted in the Red Hat "ifcfg" network configuration system by specifying a `METRIC` number in the `ifcfg-` files. This metric will be applied to any route populated by DHCP or by a `GATEWAY` value in the `ifcfg-` file or `/etc/sysconfig/network` file.

```
# grep METRIC= /etc/sysconfig/network-scripts/ifcfg-ens192
METRIC=100

# grep METRIC= /etc/sysconfig/network-scripts/ifcfg-ens224
METRIC=200
```

Alternatively, routes may be specified using `route-` files. These routes must define metrics explicitly.

```
# cat /etc/sysconfig/network-scripts/route-ens192
default via 10.225.160.1 dev ens192 metric 100

# cat /etc/sysconfig/network-scripts/route-ens224
default via 10.225.176.1 dev ens224 metric 200
```

Alternatives and Further Improvements

The NetworkManager service in RHEL 7.x handles multiple default routes correctly by supplying distinct metrics automatically; but, of course, specifying route metrics manually allows you to control which route is preferred explicitly.

I continue to wonder whether it might be better to go completely dynamic and actually run OSPF [2] on all multi-homed hosts. This should—in theory—allow our network to be even more automatically dynamic in response to link availability, but this may be too complex to justify in our environment.

There's also potential to use all available routes simultaneously with weighted load-balancing, either per-flow or per-packet [3]. This is generally inappropriate in our environment but could be preferable in an environment where the available networks are definitively general-purpose.

```
# ip route equalize add default \
    nexthop via 10.225.160.1 dev ens192 weight 1 \
    nexthop via 10.225.176.1 dev ens224 weight 10
```

Conclusion

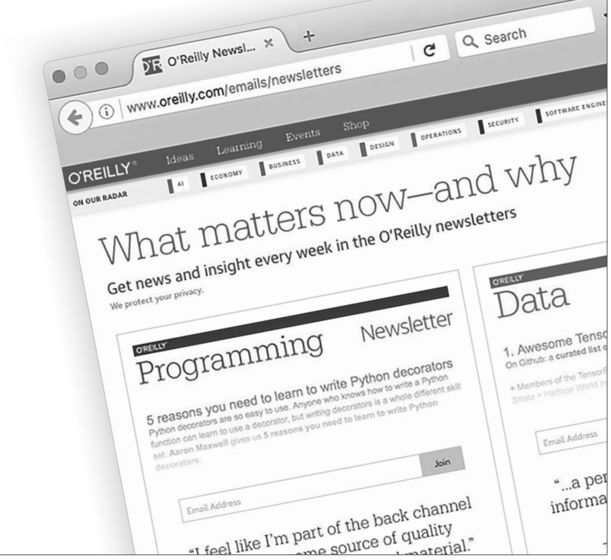
We've integrated a multiple-default-route configuration into our standard production network configuration, which is being deployed in parallel with our migration to policy routing. Now the default route is specified not by the static binary existence of a single `default` entry in the routing table but by an order of preference for each of the available interfaces. This allows our hosts to remain functional in more failure scenarios than before, when link failure or network maintenance makes the preferred route unavailable.

References

[1] iproute2: http://www.linuxfoundation.org/collaborate/workgroups/networking/iproute2.

[2] OSPFv2: https://www.ietf.org/rfc/rfc2328.txt.

[3] Policy Routing: http://www.policyrouting.org/PolicyRoutingBook/ONLINE/CH05.web.html.



What matters now—and why

Get news and insight every week in the O'Reilly newsletters

We protect your privacy.

O'REILLY Programming Newsletter

5 reasons you need to learn to write Python decorators

Python decorators are so easy to use. Anyone who knows how to write a Python function can learn to use a decorator, but writing decorators is a whole different skill set. Aaron Maxwell gives us 5 reasons you need to learn to write Python decorators.

"I feel like I'm part of the back channel ... source of quality ... material."

O'REILLY Data

1. Awesome Tenso
On Github: a curated list o

"...a per
informa

SECURITY

Bootstrapping Trust in Distributed Systems with Blockchains

MUNEEB ALI, JUDE NELSON, RYAN SHEA, AND MICHAEL J. FREEDMAN

Muneeb Ali is the co-founder and CTO of Blockstack Labs and a final-year PhD candidate at Princeton University, where he has worked in the Systems and Networks group and at PlanetLab. He helped start a new course at Princeton on "How to Be a CTO" and gives guest lectures on cloud computing at Princeton. Muneeb has been awarded a J. William Fulbright Fellowship. muneeb@blockstack.com

Jude Nelson is an Engineering Partner at Blockstack Labs and a final-year PhD candidate at Princeton University working with Larry Peterson. For over five years, Jude worked as a core member of PlanetLab. He has received nearly $4 million in research grants, and his systems have been deployed across dozens of universities. jude@blockstack.com

Ryan Shea is the co-founder and CEO of Blockstack Labs. He graduated from Princeton University, where he studied computer science and mechanical and aerospace engineering. He was also an engineer at ZocDoc and President of the Princeton Entrepreneurship Club. Recent honors include a Forbes 30 under 30 award. ryan@blockstack.com

Michael J. Freedman is a Professor of Computer Science at Princeton University, with a research focus on distributed systems, networking, and security. Recent honors include a Presidential Early Career Award (PECASE) as well as early investigator awards through the NSF and ONR, a Sloan Fellowship, and DARPA CSSG membership. mfreed@cs.princeton.edu

Blockchains like Bitcoin and Ethereum have seen significant adoption in the past few years. Beyond their cryptocurrency uses, blockchains are being used to build new, decentralized versions of DNS and public-key infrastructure (PKI) that have no central points of trust. Such blockchain-based naming and PKI services can be used as a general-purpose "trust layer" for Internet applications. We present the design of a new blockchain-based naming and storage system called Blockstack. Blockstack powers a production system for 60,000 users and is released as open source software.

Cryptocurrency blockchains and their respective P2P networks are useful beyond exchanging money. They provide cryptographically auditable, append-only global logs that have a high computational-cost barrier for tampering with data written to them. Blockchains have no central points of trust or failure: they minimize the degree to which users/nodes need to trust a single party, such as a DNS root server or a root certificate authority.

Blockchain networks have attracted a lot of interest from enthusiasts, engineers, and investors. In fact, $1.1 billion has been invested in blockchain startups over the past few years [5]. With this rapid capital infusion, infrastructure for blockchains is getting quickly deployed, and they are emerging as publicly available common infrastructure for building services and applications.

Blockchains are already being used to build new, *decentralized* versions of DNS (http://namecoin.info) and public-key infrastructure (http://onename.com) that have no central points of trust. Such blockchain-based naming and PKI services can be used as a general-purpose "trust layer" for other distributed systems and applications. For example, an IoT node can be registered on a blockchain with a unique name and controlled by a cryptographic keypair binding, stored on the blockchain, with that name.

Blockchain-based naming systems securely bind *names*, which can be human-readable, to arbitrary *values*. The blockchain gives consensus on the global state of the naming system and provides an append-only global log for state changes. Writes to name-value pairs can only be announced in new blocks, as appends to the global log. The global log is fully replicated (all nodes on the network see the same state) but organizationally decentralized (no central party controls the log).

The decentralized nature of blockchain-based naming introduces meaningful security benefits, but certain aspects of contemporary blockchains present technical limitations. Individual blockchain records are typically on the order of kilobytes [10] and cannot hold much data. The latency of creating and updating records is capped by the blockchain's write propagation and leader election protocol, and it is typically on the order of 10–40 minutes [4]. The total number of new operations in each round is limited by the average bandwidth of the network's nodes; for Bitcoin, the current average is ~1500 new operations per new round [5]. Further, new nodes need to independently audit the global log from its beginning: as the system makes forward progress, the time to bootstrap new nodes increases linearly.

We believe that in spite of these scalability and performance challenges, blockchains provide important infrastructure for bootstrapping trust in distributed systems and building new decentralized services. The cost of tampering with blockchains grows with their adoption: today, it would require hundreds of millions of dollars to attack a large blockchain like Bitcoin [5].

These benefits motivated us to use blockchains to build a new global naming and storage system, called Blockstack [3]. Our system enables users to register unique, human-readable usernames and associate public-keys, like PGP, along with additional data to these usernames. In this article, we present an overview of the design of Blockstack and discuss how it can be used as a general-purpose "trust layer" for building other applications and services. Unlike previous blockchain-based systems, Blockstack *separates its control and data plane considerations*: it keeps only minimal metadata (namely, data hashes and state transitions) in the blockchain and uses external datastores for actual bulk storage. Blockstack enables fast bootstrapping of new nodes by using checkpointing and *skip lists* to limit the set of blocks that a new node must audit to get started.

Modifying production blockchains like Bitcoin (and introducing new functionality for which it was not designed) is quite difficult, particularly since the system still needs to reach "consensus." With Blockstack, we extend the single-state machine model of blockchains to allow for arbitrary state machines without requiring consensus-breaking changes in the underlying blockchain. This design was non-intuitive before our work; indeed, the standard approach for the past three years was to fork the main Bitcoin blockchain to add new and different functionality. We have released Blockstack as open source (http://github.com/blockstack).

Motivation and Background

We next describe the motivation for building naming systems that have no central point of trust and provide the relevant background on blockchains. We use the term *naming system* to mean: (1) names are *human-readable* and can be picked by humans; (2) name-value pairs have a *strong sense of ownership*—that is, they can be owned by cryptographic keypairs; and (3) there is *no central trusted party* or point of failure. Building a naming system with these three properties was considered impossible according to Zooko's Triangle (http://dankaminsky.com/2011/01/13/spelunk-tri), and most traditional naming systems provide two out of these three properties [8]. Namecoin used a blockchain-based approach to provide the first naming system that offered all three properties: human-readability, strong ownership, and decentralization.

Background on Blockchains

Blockchains provide a global append-only log that is publicly writeable. Writes to the global log, called *transactions*, are organized as *blocks,* and each block packages multiple transactions into a single atomic write. Writing to the global log requires a payment in the form of a *transaction fee*. Nodes participating in a blockchain network follow a leader election protocol for deciding which node gets to write the next block and collect the respective transaction fees. Only one node gets to write a block in each leader election round. Not all nodes in the network participate in leader election. Those actively competing to become the leader of the next round are called *miners*. At the start of each round, all miners start working on a new computation problem, derived from the last block, and the miner that is the first to solve the problem gets to write the next block. In Bitcoin, the difficulty of these computation problems is automatically adjusted by the protocol so that one new block is produced roughly every 10 minutes. See [4] for further details on how blockchains work and how they reach consensus.

Naming System on a Blockchain

The first blockchain to implement a naming system was Namecoin. It is one of the first forks of Bitcoin and is the oldest blockchain other than Bitcoin that is still operational. The main motivation for starting Namecoin was to create an alternate DNS-like system that replaces DNS root servers with a blockchain for mapping domain names to DNS records [8]. Given that blockchains don't have central points of trust, a blockchain-based DNS is much harder to censor, and registered names cannot be seized from owners without getting access to their respective private keys [8]. Altering name registrations stored in a blockchain requires prohibitively high computing resources because rewriting blockchain data requires *proof-of-work* [2]. Before our work, it was common practice to start new blockchains (by forking them from Bitcoin) to introduce new functionality and make modifications required by the respective service/application, which is the precise approach taken by Namecoin.

Just like DNS, there is a cost associated with registering a new name. The name registration fee discourages people from registering a lot of names that they don't actually intend to use. In Namecoin, the recipient of registration fees is a "black hole" cryptographic address from which money cannot be retrieved [8]. Namecoin defines a pricing function for how the cost of name registrations changes over time. Namecoin supports multiple namespaces (like TLDs in DNS), and the same rules for pricing and name expiration apply to all namespaces. By convention, the *d/* namespace is used for domain names.

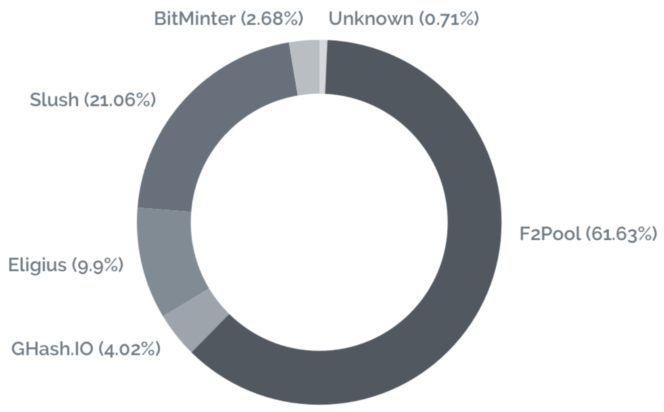


Figure 1: Mining distribution for Namecoin (Aug. '15)

In Namecoin, name registration uses a two-phase commit method where a user first *pre-orders* a name hash and then *registers* the name-value pair by revealing the actual name and the associated value. This is done to avoid front-running unconfirmed name registrations [8]. Name registrations expire after a fixed amount of time, measured in new blocks written (currently 36,000 blocks, which translates to roughly eight months), and cannot be purchased for longer periods. Namecoin also supports updating the value associated with a name, as well as ownership transfers.

Our experience with the Namecoin blockchain shows that starting new, smaller blockchains leads to security problems (like reduced computational power needed to attack the network) and should be avoided when possible. We detailed our findings in a recent USENIX ATC paper [3]. *Importantly, we discovered a critical security problem where a single miner consistently had more than 51% of the total compute power on the Namecoin network* (see Figure 1 for the Namecoin mining distribution for the month of August 2015, and see [4] for details on the 51% attack and compute power of miners). A 51% attack is one of the most serious attacks on a blockchain and impacts its security and decentralization properties. Other than Namecoin, blockchains like Ethereum [1] and BitShares (http://bitshares.org) also have support for human-readable names (tied to their respective blockchains).

Design of Blockstack

Blockstack is designed to implement a naming system with human-readable names in a layer above the blockchain. In this section, we describe how Blockstack uses the underlying blockchain and present how it copes with technical limitations of contemporary blockchains.

Challenges

Building systems with blockchains presents challenges:

- **Limits on data storage:** Individual blockchain records are typically on the order of kilobytes [10] and cannot hold much data. Moreover, the blockchain's log structure implies that *all* state changes are recorded in the blockchain. All nodes participating in the network need to maintain a full copy of the blockchain, limiting the total size of blockchains to what current commodity hardware can support. As of May 2016, Bitcoin nodes need to dedicate 69 GB total disk space to blockchain data for staying synchronized with the network.
- **Slow writes:** The transaction processing rate is capped by the blockchain's write propagation and leader election protocol, and it is pegged to the rate at which new blocks are announced by leader nodes (miners). New transactions can take several minutes to a few hours to be accepted.
- **Limited bandwidth:** The total number of transactions per block is limited by the *block size* of blockchains. To maintain fairness and to give all nodes a chance to become leader in the next round, all nodes should receive a newly announced block at roughly the same time. Therefore, the block size is typically limited by the average uplink bandwidth of nodes. For Bitcoin, the current block size is 1 MB (~1000 transactions).
- **Endless ledger:** The integrity of blockchains depends on the ability of anyone to audit them back to their first block. As the system makes forward progress and issues new blocks, the cost of an audit grows linearly with time. Thus, booting up new nodes becomes progressively more time-consuming. We call this the *endless ledger problem*. As of May 2016, Bitcoin's blockchain had ~413,000 blocks, and new nodes take 1–3 days to download the blockchain from Bitcoin peers, verify it, and boot up.

Architecture Overview

Blockstack maintains a naming system as a separate logical layer on top of the underlying blockchain on which it operates. Blockstack uses the underlying blockchain to achieve consensus on the state of this naming system and bind names to data records. Specifically, it uses the underlying blockchain as a communication channel for announcing state changes, as any changes to the state of name-value pairs can only be announced in new blockchain blocks. Relying on the consensus protocol of the underlying blockchain, Blockstack can provide a total ordering for all operations supported by the naming system, like name registrations, updates, and transfers.

Separation of the Control and Data Plane: Blockstack decouples the security of name registration and name ownership from the availability of data associated with names by separating the control and data planes.

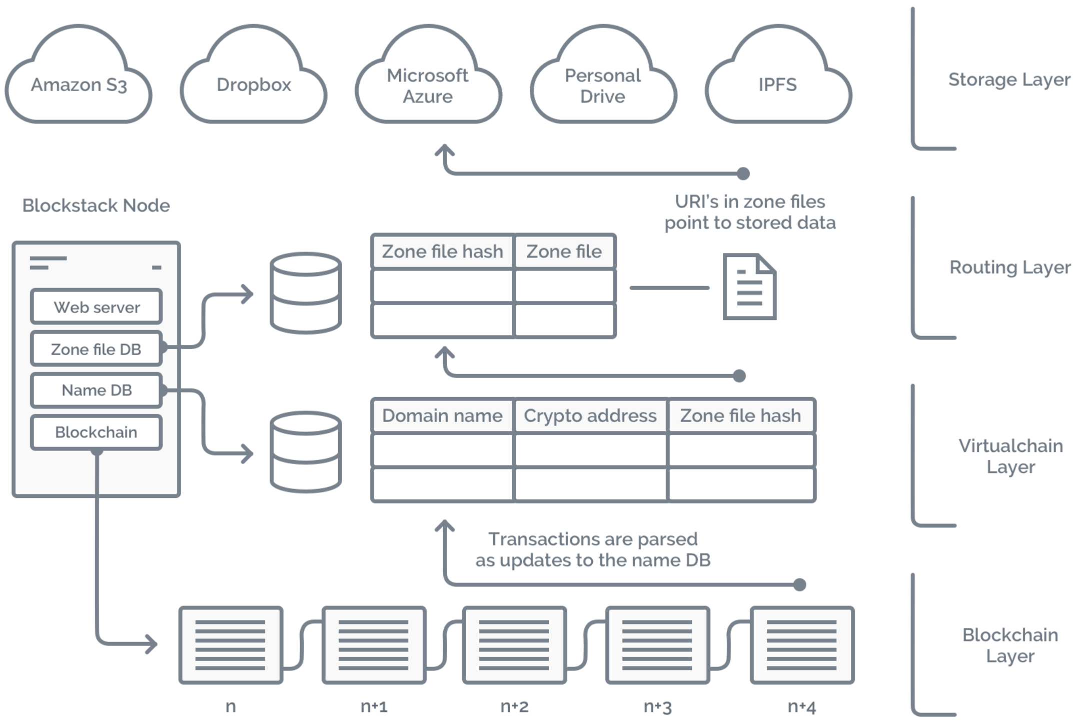


Figure 2: Overview of Blockstack's architecture. Blockchain records give (name, hash) mappings. Hashes are looked up in routing layer to discover routes to data. Data, signed by name owner's public-key, is stored in cloud storage.

The control plane defines the protocol for registering human-readable names, creating (*name,hash*) bindings and mapping name ownership to cryptographic key pairs. The control plane consists of a blockchain and a logically separate layer on top, called a *virtualchain*.

The data plane is responsible for data storage and availability. It consists of (1) zone files for discovering data by hash or URL and (2) external storage systems for storing data (such as S3, IPFS [6], and Syndicate [7]). Data values are signed by the public keys of the respective name owners. Clients read data values from the data plane and verify their authenticity by checking that either the data's hash is in the zone file, or the data includes a signature with the name owner's public key.

We believe this separation is a significant improvement over Namecoin, which implements both the control and the data plane at the blockchain level. Our design not only significantly increases the data storage capacity of the system, but also allows each layer to evolve and improve independently of the other.

Agnostic of the Underlying Blockchain: The design of Blockstack does not put any limitations on which blockchain can be used with it. Any blockchain can be used, but the security and reliability properties are directly dependent on the underlying blockchain. We believe that the ability to *migrate* from one blockchain to another is an important design choice as it allows for the larger system to survive, even when the underlying blockchain is compromised. Currently, Blockstack core developers decide which underlying blockchain(s) to support in which version of the software. Individual applications can decide to run the software version of their choice and keep their namespace on a particular blockchain, if they prefer not to migrate.

Ability to Construct State Machines: A key contribution of Blockstack is the introduction of a logically separate layer on top of a blockchain that can construct an arbitrary *state machine* after processing information from the underlying blockchain. This virtualchain treats transactions from the underlying blockchain as inputs to the state machine; valid inputs trigger state changes. At any given time, where time is defined logically by the block number, the state machine can be in exactly one global state. Time moves forward as new blocks are written in the underlying blockchain, and the global state is updated correspondingly.

A virtualchain enables the introduction of new types of state machines, without requiring any changes to the underlying blockchain. This approach is especially beneficial for a new and developing technology. Introducing new state machines directly in a blockchain would otherwise require peers to upgrade, and upgrades potentially break consensus and cause forks. In practice, they are difficult to orchestrate [4]. Currently, Blockstack provides a state machine that represents the global state of a naming system, including who owns a particular name and what data is associated with a name. Further, it's possible to use the virtualchain concept to define other types of state machines as well.

Blockstack Layers

Blockstack introduces new functionality on top of blockchains by defining a set of new operations that are otherwise not supported by the blockchain. Blockstack has four layers, with two layers (blockchain and virtualchain) in the control plane and two layers (routing and data storage) in the data plane.

Layer 1: Blockchain Layer

The blockchain occupies the lowest tier (see Figure 2) and serves two purposes: it stores the sequence of Blockstack operations, and it provides consensus on the order in which the operations are written. Blockstack operations are encoded in transactions on the underlying blockchain.

Layer 2: Virtualchain Layer

Above the blockchain is a virtualchain, which defines new operations without requiring changes to the underlying blockchain. Only Blockstack nodes are aware of this layer, and underlying blockchain nodes are agnostic to it. Blockstack operations are defined in the virtualchain layer and are encoded in valid blockchain transactions as additional metadata. Blockchain nodes do see the raw transactions, but the logic to process Blockstack operations only exists at the virtualchain level.

The rules for accepting or rejecting Blockstack operations are also defined in the virtualchain. Accepted operations—for example, a name registration operation on a name that has not been registered by anyone yet—are processed by the virtualchain to construct a database that stores information on the current global state of the system along with a history of previous states at earlier blockchain blocks.

Layer 3: Routing Layer

Blockstack separates the task of *routing requests* (i.e., how to discover data) from the actual storage of data. This avoids the need for the system to adopt any particular storage service from the outset and instead allows multiple storage providers to coexist, including both commercial cloud storage and peer-to-peer systems.

Blockstack uses *zone files* for storing routing information, which are identical to DNS zone files in their format. Figure 3 shows an example zone file. The virtualchain binds names to their respective *hash(zone file)*. While these bindings are stored in the control plane, the zone files themselves are stored in the routing layer. *Users do not need to trust the routing layer;* this is because, as long as the zone file data is available, the integrity of zone files can be verified by checking their hash from the control plane.

```
$ORIGIN werner.id
$TTL 3600
_http._tcp URI 10 1 http://54.231.237.47/werner.id
```

Figure 3: Example zone file

Currently, all zone files are public and globally resolvable, and we plan to support private data linked to public zone files in the future.

In Blockstack's current implementation, nodes form a DHT-based peer network [9] for storing zone files. The DHT (distributed hash table) only stores zone files if their hash was previously announced in the blockchain. This effectively whitelists the data that can be stored in the DHT. Due to space constraints, we omit most details of our DHT storage from this article; the key aspect relevant to the design of Blockstack is that routes (irrespective of where they are fetched from) can be verified and therefore cannot be tampered with. Further, most production servers maintain a full copy of all zone files since the size of zone files is relatively small (4 KB per file). Keeping a full copy of routing data introduces only a marginal storage cost (24 MB as of June 2016) on top of storing the blockchain data.

Layer 4: Storage Layer

The topmost layer is the storage layer, which hosts the actual data values of name-value pairs. All stored data values are signed by the key of the respective owner of a name. By storing data values outside of the blockchain, Blockstack allows values of arbitrary size and allows for a variety of storage backends. *Users do not need to trust the storage layer:* as long as the zone file data is available, they can verify the integrity of the data values in the control plane.

There are two modes of using the storage layer, and they differ in how the integrity of data values is verified; Blockstack supports both storage modes simultaneously.

Mutable storage is the default mode of operation for the storage layer. The user's zone file contains a URI record that points to the data, and the data is constructed to include a signature from the user's private key. Writing the data involves signing and replicating the data (but not the zone file), and reading the data involves fetching the zone file and data, verifying that hash(zone file) matches the hash in Blockstack, and verifying the data's signature with the user's public key. This allows for writes to be as fast as the signature algorithm and underlying storage system allows, since updating the data does not alter the zone file and thus does not require any blockchain transactions. However, readers and writers must employ a data versioning scheme to avoid consuming stale data.

Immutable storage is similar to mutable storage, but additionally puts a TXT record in the zone file that contains *hash(data)*. Readers verify data integrity by fetching the data and checking that hash(data) is in the zone file, in addition to verifying the data's signature and the zone file's authenticity. This mode is suitable for data values that don't change often and where it's important to verify that readers see the latest version of the data value. For immutable storage, updates to data values require a new transaction on the underlying blockchain (since the zone file must be modified to include the new hash), making data updates much slower than with mutable storage.

Naming System

Blockstack uses its four tiers to implement a complete naming system. Names are owned by cryptographic addresses of the underlying blockchain and their associated private keys (e.g., ECDSA-based private keys used in Bitcoin [4]). As with Namecoin, a user *preorders* and then *registers* a name in two steps in order to claim a name without revealing it to the world first and allowing an attacker to race the user in claiming the name. The first user to successfully write both a preorder and a register transaction is granted ownership of the name. Further, any previous preorders become invalid when a name is registered. Once a name is registered, a user can *update* the name-value pair by sending an update transaction and uploading the new value to the storage layer, changing the name-value binding. Name *transfer* operations simply change the address that is allowed to sign subsequent transactions, while *revoke* operations disable any further operations for names.

The naming system is implemented by defining a state machine and rules for state transitions in the virtualchain. Names are organized into *namespaces*, which are the functional equivalent of top-level domains in DNS—they define the costs and renewal rates of names. Like names, namespaces must be preordered and then registered. Expired names can be re-registered, and names can be *revoked* such that they cannot be re-registered for a certain period of time.

Pricing Functions for Namespaces

Anyone can create a namespace or register names in a namespace since there is no central party to stop someone from doing so. *Pricing functions* define how expensive it is to create a namespace or to register names in a namespace. Defining intelligent pricing functions is a way to prevent "land grabs" and stop people from registering a lot of namespaces or names that they don't intend to actually use. Blockstack enables people to create namespaces with sophisticated pricing functions. For example, we use the *.id* namespace for our PKI system and created the *.id* namespace with a pricing function where (1) the price of a name drops with an increase in name length and (2) introducing non-alphabetic characters in names also drops the price. With this pricing function, the price of *john.id* > *johnadam.id* > *john0001.id*. The function is generally inspired by the observation that short names with alphabetic characters only are considered more desirable on namespaces like the one for Twitter usernames. It's possible to create namespaces where name registrations are free as well. Further, we expect that in the future there will be a reseller market for names, just as there is for DNS. A detailed discussion of pricing functions is beyond the scope of this article, and the reader is encouraged to see [8] for more details on pricing functions.

Like names, namespaces also have a *pricing function* [3]. *To start the first namespace on Blockstack, the .id namespace, we paid $10,000 in bitcoins to the network. This shows that even the developers of this decentralized system have to follow Blockstack rules and pay appropriate fees.*

Conclusion

We have presented Blockstack, a blockchain-based naming and storage system that can be used as a general-purpose "trust layer" for building other service and applications without relying on any third parties. Blockstack introduces separate control and data planes and, by doing so, enables the introduction of new functionality without modifying the underlying blockchain. This is counter to prior designs, which typically involved the introduction of a new blockchain and cryptocurrency in order to introduce new functionality.

The design of Blockstack was informed by a year of production experience from one of the largest blockchain-based production systems to date. We have introduced several innovations for blockchain services, including the ability to do cross-chain migrations, faster bootstrapping of new nodes, and keeping data updates off the slow blockchain network. These improvements all make it easier to build new decentralized services using existing, publicly available blockchain infrastructure.

References

[1] Ethereum Wiki, "A Next-Generation Smart Contract and Decentralized Application Platform," 2016: https://github.com/ethereum/wiki/wiki/White-Paper.

[2] A. Back, "Hashcash—A Denial of Service Counter-Measure," Tech Report, 2002: http://www.hashcash.org/papers/hashcash.pdf.

[3] M. Ali, J. Nelson, R. Shea, and M. Freedman, "Blockstack: A Global Naming and Storage System Secured by Blockchains," in *Proceedings of the 2016 USENIX Annual Technical Conference (ATC '16)*, June 2016.

[4] J. Bonneau, A. Miller, J. Clark, A. Narayanan, J. A. Kroll, and E. W. Felten, "Sok: Research Perspectives and Challenges for Bitcoin and Cryptocurrencies," in *2015 IEEE Symposium on Security and Privacy (SP 2015)*, pp. 104–121.

[5] CoinDesk, "State of Blockchain Q1 2016: Blockchair Funding Overtakes Bitcoin," May 2016: http://www.coindesk.com/state-of-blockchain-q1-2016/.

[6] J. Benet, "IPFS—Content Addressed, Versioned, P2P File System," Draft, ipfs.io, 2015: https://github.com/ipfs/papers.

[7] J. Nelson and L. Peterson, "Syndicate: Virtual Cloud Storage through Provider Composition," in *Proceedings of the 2014 ACM HPDC International Workshop on Software-Defined Ecosystems (BigSystem '14)*.

[8] H. Kalodner, M. Carlsten, P. Ellenbogen, J. Bonneau, and A. Narayanan, "An Empirical Study of Namecoin and Lessons for Decentralized Namespace Design," in *Proceedings of the 14th Workshop on the Economics of Information Security (WEIS '15)*, June 2015.

[9] P. Maymounkov and D. Mazières, "Kademlia: A Peer-to-Peer Information System Based on the Xor Metric," in *Revised Papers from IPTPS (IPTPS '01)*, 2002, pp. 53–65.

[10] S. Nakamoto, "Bitcoin: A Peer-To-Peer Electronic Cash System," Tech Report, 2009: https://bitcoin.org/bitcoin.pdf.

Practical Threat Modeling

BRUCE POTTER

Bruce Potter, CTO of the KEYW Corporation, has over 20 years of experience focused on tackling high-end information security research and engineering problems. During his career, Bruce has built and led teams focused on hard problems in information security such as cybersecurity risk analysis, high assurance system and network engineering, advanced software analysis techniques, wireless security, and IT operations best practices. Bruce is also the founder of The Shmoo Group, a nonprofit think tank comprising security, privacy, and crypto professionals who donate time to information security research and development. Bruce assists in the organization of ShmooCon, an annual computer security conference in Washington, D.C. Bruce has authored many publications and has delivered numerous presentations at various security and network conferences and private events, including DefCon, BlackHat USA, ShmooCon, the United States Military Academy, Johns Hopkins University, and the Library of Congress. bpotter@keywcorp.com

Threat modeling is a key component to modern-day cybersecurity risk management, but the processes of creating a threat model can be complex and overwhelming. By understanding the components that make up statements of threat, such as threat actors, assets, and malicious actions, we can turn threat modeling into a management process that can be performed by a variety of practitioners. With some practice and awareness of your enterprise, you can start creating threat models that will have a large impact on the quality of your risk management decisions.

Information security is really all about risk management. Building provably secure systems is cost prohibitive and serves as a barrier to innovation. Most modern-day systems not only operate in a constantly changing environment but are incredibly complex and diverse collections of hardware and software with many interfaces and numerous use cases.

Rather than secure all parts of a system equally, we must invest our time and resources wisely to secure a system in the places that actually matter. The idea of addressing the areas of highest concern first is really the core concept behind risk management. Without understanding the risk to the system, we might as well roll dice to determine what to focus our efforts on.

Why Is Threat Modeling Important?

There are several ways to think about risk. While distilling risk down to a simple equation has some dangers associated with it, the core concepts behind a risk equation are a useful foil to discuss risk. For our purposes, assume understanding a risk follows the equation below:

Risk = ((Threat x Vulnerability) / Countermeasure) x Impact

Understanding each of the values that go into this equation is its own discipline. For instance, to understand vulnerabilities in a system, you might employ product evaluators or penetration testers who will examine your system from top to bottom to find security vulnerabilities and document them. To understand countermeasures, you could perform a security control audit to find out where all your security controls are, if they are configured correctly, and how effective they are. Finally, to examine impact, you might meet with various business managers within your organization to better understand how critical each system is and the overall value to the business.

Understanding the threat has been a more elusive problem. The concept of threat feels like it has more basis in intelligence gathering than technical analysis. The idea of someone in an organized crime ring sitting in a dark room halfway across the world writing custom malware targeting your Web site sounds like something out of a spy movie. But understanding the threats facing your systems doesn't require you to hire a security intelligence service and go deep underground to find all the organizations wishing to do you harm.

Threat modeling is a process that is used to develop and rank specific threats against your system. The resulting threat model is a document with a similar audience as a technical risk or vulnerability assessment. The threat model can be used by developers to understand what attackers might try to do to the system when they are determining how to code defensively. The threat model can be used by system and network operators to help determine what network controls should be put in place based on potential adversarial actors. The model can even be used by management to assist in understanding the threat landscape and adjust development and IT spend.

The process of threat modeling can be very complex. Microsoft, as part of its well documented and publicly available security development life cycle, published a book [1] that documents their threat modeling process. Adam Shostack's *Threat Modeling: Designing for Security* puts forth a great process for threat modeling in the development process. The book is comprehensive and can be applied in structured development environments. However, the process described in *Threat Modeling* can be very heavyweight, especially in lean or understaffed development environments. And the process is very difficult to modify for suitability for use for non-developers. The threat modeling process put forth in this article has been influenced by a number of sources, including *Threat Modeling*. However, I have created this process to be useful and applicable to a broad audience of practitioners, not just system developers.

All that said, what exactly is a threat?

The Syntax of a Threat

In the context of threat modeling, I've found it is useful to think about threats using a very specific syntax:

$ACTOR does $ACTION to $ASSET for $OUTCOME because $MOTIVATION

A threat model is a collection of threat statements that follow this basic syntax listed in ranked order. In order to create a specific threat in a threat model, we must have specific knowledge about an instance of each variable. The process of creating a threat model involves identifying interesting values for each variable and determining which are important to your organization.

The first three variables ($actor, $action, and $asset) are somewhat self-explanatory and will be covered later in this article. The outcome is critically important to each threat. There are many actions that an adversary could take involving your system. However, if the action results in no bad outcome, there is no consequence to the action. For instance, if an attacker can anonymously log in to your FTP server but the only data on the server is public data, then there's no bad outcome. In fact, your adversary is acting with the same privilege and access as your regular users. Anonymous FTP access to your FTP server in this case is not a threat action.

The last variable, motivation, is somewhat optional. The motivation of an adversary is not necessarily of interest to every organization. Some organizations are interested in what is motivating their attackers and use that information to develop deterrence strategies. Other organizations do not have as robust an understanding of adversaries and only care about the outcome, not the motivation. As you go through this process, you will get a sense of what you and your organization cares about and can decide whether capturing the attacker's motivation is important to you.

Threat Actors

While there are specific bad actors in the world that may wish to harm your systems, you don't necessarily need to identify them by name. Rather, thinking of threat actors in broad categories helps you understand motivations and resourcing and how that would impact what they can and would do. The following five major threat actor categories are a useful starting point for you to develop an understanding of threat actors and the role they will have in your model.

Nation State

Nation state actors are very well resourced and may maintain operations for months or even years. These actors are motivated by national interests such as intelligence gathering, military action, critical infrastructure control, and industrial espionage. Nation states are generally very difficult to defend against.

Organized Crime

Organized crime actors are moderately well resourced with operations that may last for months. These actors are generally motivated by financial gain or access to information that can lead to financial gain such as personal information or credit card data. Due to the focused nature of organized crime, they can be very difficult to defend against, although the information they are interested in is often more limited than that of nation states.

Insiders

Insiders are as well resourced as you let them be. Insiders will utilize whatever access is available to accomplish their objective. Given the state of internal security of most organizations, insiders are far over-accessed and can cause great harm. Motivation for insiders can range from ideological issues to profit to revenge. Insiders are difficult to defend against as they may dedicate their lives to pursuing their objective.

Hacktivists

Hacktivists have limited resources and run operations that last weeks to months. They are generally motivated by ideological issues and target organizations very specifically. Hacktivists often publicly discuss their objectives and hide behind anonymity services such as Tor. Organizations with well-run IT operations can often defend against hacktivists, although social engineering can be the Achilles heel of enterprises under hacktivist attack.

Script Kiddies

Script kiddies have very limited resources. These adversaries are often motivated by curiosity and simple malicious intent. Their tooling often only consists of publicly available tools, hence the "script kiddie" moniker. These attackers will look for targets of opportunity and can be defended against using normal IT security best practices.

Others

As you work through several threat models, you may find that you identify specific threat actors that are unique to your organization. Maybe you have been targeted by an organized crime group in the past with particular interest in your company. Or maybe Bob from Accounting seems like he might be up to no good. Whatever the reason, feel free to add to the list of threat actors as your models evolve. However, be cautious of creating too many specific actors; if their resourcing and motivations are similar, there's little utility in splitting out multiple actors.

System Representation

The first step in the threat modeling process is to create a system representation. In Microsoft's process, the system is represented formally through Data Flow Diagrams (DFD). These DFDs capture all interfaces, assets, and data flows through very specific iconography. While Microsoft DFDs are in some contexts a universal language, they are also time-consuming to create and at such a low level to not be useful to non-developers.

Rather than completely decomposing the system you are threat modeling, capture the system in a manner that is convenient for you and your team. These might be network diagrams, system architectures, or even basic scribbling on a whiteboard. What's important in capturing the representation is that it captures the assets and capabilities you are trying to protect. For instance, if you are threat modeling a CRM (customer relationship management) solution, your system representation should include your sales force, the types of systems they use to access the solution, transport and storage mechanisms, the CRM servers themselves, and any external data sources. Whether the servers are circles or squares really doesn't matter; what does matter is that all the components of the system are represented. Ultimately, these diagrams capture all the $assets that you will use to create the threat model.

Brainstorming

The real meat of the threat modeling process is brainstorming. This is the part of the process that requires a little bit of creativity and security knowledge. Start with an asset represented in your system representation. Pick a threat actor and then think of the bad things the attacker may do to that asset and how that would affect your organization. Write down that threat in the syntax described above, and then think of another bad thing that threat actor could do…and another…and another. Write down ideas for that threat actor until you've run out of ideas, then go to another threat actor.

It may seem like you could just write a program that does something like

```
Foreach (ASSET)
        Foreach (ACTOR)
                Foreach (ACTION)
                        Print "$ACTOR does $ACTION to $ASSET
```

and BOOM you'd have a threat model. While this is true, in reality the number of threats you would come up with is astronomical. This is where common sense comes in to play. The idea of a script kiddie launching a highly sophisticated attack involving a large amount of resources is nonsensical. Similarly, there are numerous attacks a nation state wouldn't carry out because they aren't motivated to, such as defacements and social media attacks. There are several techniques you can use to help target your threat statements.

Use Your Knowledge

Only write down threats that you think are real issues. There's a great deal of knowledge of contemporary attack techniques and motivations. Based on the line of work that your business is engaged in, the specifics of your assets, and the capabilities of various threat actors, use your knowledge of security and attacks to capture threats that make sense. For instance, if you run a large retail operation: we know that in the attack against Target and other institutions, attackers went after the point-of-sale terminals. Therefore, when thinking of threats against point-of-sale systems, a threat like this is appropriate:

- Organized crime group places RAM scraper on point-of-sale terminals in order to steal mag stripe data to facilitate fraud

You'll note that this threat adheres to our threat syntax

- Organized crime group ($actor) uses physical access to place RAM scraper ($action) on point-of-sale terminals ($asset) in order to steal mag stripe data ($outcome) to facilitate fraud ($motivation)

This threat is contemporary and is likely of high concern to retail organizations.

A Threat Is Specific as It Needs to Be

Sometimes a threat does not need all the syntactical parts in order to be useful. Take potential threats against a network router. As we brainstorm what different threat actors would do, we might find that we end up with a list of threats such as these:

- **Nation state** performs denial of service on **router** to stop all access to internal services from Internet
- **Organized crime** performs denial of service on **router** to stop all access to internal services from Internet
- **Insider** performs denial of service on **router** to stop all access to internal services from Internet

Note that multiple threat actors may do the exact same action to your router. While the motivation may be different, the actual attack is the same. Unless you are really worried about the motivation, you can distill these three threats into one:

- Network-based denial of service against router stops all access to internal services from Internet

This threat is still actionable even though a specific threat actor is not represented.

Taking a Break

Brainstorming threats is tough work. It can be difficult to be creative in developing a long list of threats during the early stages. Work your way through all the components in the system representation in the first pass, then take a break. Go do something else, take a walk, drink a beer, or just go home for the day. Whatever you need to do to get away from creating threats, do it. From our experience, taking three or four cycles on adding threats to the list is when we hit the point of diminishing returns. After several sessions of listing threats, you've generally run through all the knowledge you have of a system and likely attack scenarios. Any more sessions will have very little impact on the quality of the resulting model.

Cutting Down the List

After several brainstorming sessions, you should have a list of between 50 and 200 threats, depending on the size of the system under analysis and the types of interfaces it has. A list that big is unusable. To be practical, a threat model should consist of between five and 20 top-line threats. These threats are the top threats the system faces and can be kept in your head as you build and operate the system. Twenty threats can be printed on a single piece of paper and taped on the office wall of every developer and operator in an organization to remind them of what they're defending against.

The first thing to do is to look through the list for threats that are just not realistic. In the process of brainstorming, we should allow ourselves creative license to dream and imagine all manner of bad actions an adversary might take. That process can sometimes lead us to strange places and result in threats that are really just nonsense. Remove these threats from the list. Mechanically, this doesn't mean deleting them. Rather, put them somewhere else. As you threat model more and more systems, having a list of threats to look back on to jar your memory is important. So never delete threats, just move them to a different document or tab.

The next thing to do with the list is to order by variables (actor, action, asset, etc.) to see if there is a way to distill multiple threats into a single threat using the idea that a "threat is as specific as it needs to be." If there are numerous threats that look like basically the same thing, spend time deciding whether they really are different or whether they have the same implication on the enterprise and can be condensed into something similar.

Finally, once you have nonsense threats removed and similar threats condensed, you can start sorting through the threats and ranking them. Ranking does not need to be a formal process; rather, you can use the basic calculus of the risk equation at the beginning of this article to think about what countermeasures you have in place, what the impact of the attack would be, and how likely the threat actor is to carry out the attack. Take your time and play around with the ordering until you think it's correct.

Once the threats are in order, look for a "cut line." Find a place in the list where threats above the list are likely to be important to developers and operators in your organization and below where folks are unlikely to care. There is no right size for the list of threats above the cut line, but generally the list of important threats should fit on a single piece of paper (printing in 6-point font doesn't count). Once you have found your cut line, take those threats, print them out, and tape them to your wall. Congratulations, you have made your first threat model!

Using the Threat Model

Now what? Unlike a risk assessment or vulnerability assessment, threat models tend to change slowly over time. As vulnerabilities are patched, a vulnerability assessment loses its utility. Vulnerability assessments may have a useful life of a few weeks. Risk assessments, which tend to focus on higher-level issues, may have a lifetime of a few months. Threat models can often live on for a year or two without any changes. Threats to an organization change slowly over time as economic and political systems evolve. After a year, you should reexamine your threat model to determine whether it needs updating; if your organization's situation is generally the same as it was, you can let it ride.

The threat model should be presented to developers and operators throughout your organization. Developers can use the model to help them write more defensible code. Operators can use the model to help implement and configure better security controls. The threat model can serve as the foundation of future risk assessments and help penetration testers understand what you are really concerned about. The threat model is a foundational piece of a risk-based approach to cybersecurity.

Threat modeling is still an art form. It is an important part of any cybersecurity program, but performing threat modeling is not a well understood process. The process and ideas put forth here are guidelines; they are not meant to be the hard and fast method you must use to create a threat model. Rather, they are a starting point. As you work through the process a few times, you will find ways to optimize and customize it to have more utility for you and your organization. Use these customizations to create even better and more relevant threat models to help secure your systems. Further, share your improvements with those around you so that we can all learn as we advance the discipline of threat modeling.

Reference
[1] A. Shostack, *Threat Modeling: Designing for Security* (Wiley, 2014).

COLUMNS

The Networks of Reinvention

DAVID BEAZLEY

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

Network programming has been a part of Python since its earliest days. Not only are there a wide variety of standard library modules ranging from low-level socket programming to various aspects of Web programming, there are a large number of third-party packages that simplify various tasks or provide different kinds of I/O models. As the network evolves and new standards emerge, though, one can't help but wonder whether the existing set of networking libraries are up to the task. For example, if you start to look at technologies such as websockets, HTTP/2, or coroutines, the whole picture starts to get rather fuzzy. Is there any room for innovation in this space? Or must one be resigned to legacy approaches from its past. In this article, I spend some time exploring some projects that are at the edge of Python networking. This includes my own Curio project as well as some protocol implementation libraries, including hyper-h2 and h11.

Introduction

For the past year, I've been working on a side project, Curio, that implements asynchronous I/O in Python using coroutines and the newfangled `async` and `await` syntax added in Python 3.5 [1]. Curio allows you to write low-level network servers almost exactly like you would with threads. Here is an example of a simple TCP Echo server:

```
from curio import run, spawn
from curio.socket import *

async def echo_server(address):
    sock = socket(AF_INET, SOCK_STREAM)
    sock.setsockopt(SOL_SOCKET, SO_REUSEADDR, True)
    sock.bind(address)
    sock.listen(1)
    while True:
        client, addr = await sock.accept()
        print('Connection from', addr)
        await spawn(echo_handler(client))

async def echo_handler(client):
    async with client:
        while True:
            data = await client.recv(100000)
            if not data:
                break
            await client.sendall(data)
    print('Connection closed')

if __name__ == '__main__':
    run(echo_server(('',25000)))
```

If you haven't been looking at Python 3 recently, there is a certain risk that this example will shatter your head. Functions defined with `async` are coroutines. Coroutines can be called from other coroutines using the special `await` qualifier. Admittedly, it looks a little funny, but if you run the above code, you'll find that it has no problem serving up thousands of concurrent connections even though threads are nowhere to be found. You'll also find it to be rather fast. This article isn't about the details of my pet project though.

Here's the real issue—even though I've written an experimental networking library, where do I take it from here? Most developers don't want to program directly with sockets. They want to use higher-level protocols such as HTTP. However, how do I support that? Because of the use of coroutines, `async`, and `await`, none of Python's standard libraries are going to work (coroutines are "all-in"—to use them they must be used everywhere). I could look at code in third-party libraries such as Twisted, Tornado, or Gevent, but they each implement HTTP in their own way that can't be applied to my problem in isolation. I'm left with few choices except to reimplement HTTP from scratch—and that's probably the last thing the world needs. Another custom implementation of HTTP, that is.

It turns out that other developers in the Python world have been pondering such problems. For example, how are all of the different networking libraries and frameworks going to go about supporting the new HTTP/2 protocol? There is no support for this in the standard library, and the protocol itself is significantly more complicated than HTTP/1.1. Is every library going to reimplement the protocol from scratch? If so, how many thousands of hours are going to be wasted sorting out all of the bugs and weird corner cases? How many developers even understand HTTP/2 well enough to do it? I am not one of those developers.

At PyCon 2016, Cory Benfield gave a talk, "Building Protocol Libraries the Right Way," in which he outlined the central problem with I/O libraries [2]. In a nutshell, these libraries mix I/O and protocol parsing together in a way that makes it nearly impossible for anyone to reuse code. As a result, everyone ends up reinventing the wheel. It causes a lot of problems as everyone makes the same mistakes, and there is little opportunity to reap the rewards of having a shared effort. To break out of that cycle, an alternative approach is to decouple protocol parsing entirely from I/O. Cory has done just that with the hyper-h2 project for HTTP/2 [3]. Nathaniel Smith, inspired by the idea, has taken a similar approach for the h11 library, which provides a standalone HTTP/1.1 protocol implementation [4].

The idea of decoupling network protocols from I/O is interesting. It's something that could have huge benefits for Python development down the road. However, it's also pretty experimental. Given the experimental nature of my own Curio project, I thought it might be interesting to put some of these ideas about protocols to the test to see whether they can work in practice. Admittedly, this is a bit self-serving, but Curio has the distinct advantage of being incompatible with everything. There is no other option than going it alone—so maybe these protocol libraries can make life a whole heck of a lot easier. Let's find out.

Reinventing Requests

As a test, my end goal is to reinvent a tiny portion of the popular requests library so that it works with coroutines [5]. Requests is a great way to fetch data from the Web, but it only works in a purely synchronous manner. For example, if I wanted to download a Web page, I'd do this:

```
>>> import requests
>>> r = requests.get('https://httpbin.org')
>>> r.status_code
200
>>> r.headers
{'connection': 'keep-alive', 'content-type': 'text/html;
charset=utf-8', 'access-control-allow-origin': '*',
'access-control-allow-credentials': 'true',
 'server': 'nginx', 'content-length': '12150',
  'date': 'Tue, 28 Jun 2016 14:58:04 GMT'}
>>> r.content
b'<!DOCTYPE html>\n<html>\n<head>\n  ...'
>>>
```

Under the covers, requests uses the `urllib3` library for HTTP handling [6]. To adapt requests to coroutines, one approach might be to port it and `urllib3` to coroutines—a task that would involve identifying and changing every line of code related to I/O. It's probably not an impossible task, but it would require a lot of work. Let's not do that.

Understanding the Problem

The core of the problem is that existing I/O libraries mix protocol parsing and I/O together. For example, if I wanted to establish an HTTP client connection, I could certainly use the built-in `http.client` library like this:

```
>>> import http.client
>>> c = http.client.HTTPConnection('httpbin.org')
>>> c.request('GET', '/')
>>> r = c.getresponse()
>>> r.status
200
>>> data = r.read()
>>>
```

However, this code performs both the task of managing the protocol and the underlying socket I/O. There is no way to replace the I/O with something different or to extract just the code related to the protocol itself. I'm stuck.

Using a Protocol Parser

Let's look at the problem in a different way using the h11 module. First, you'll want to install h11 directly from its GitHub page [4]. h11 is an HTTP/1.1 protocol parser. It performs no I/O itself. Let's play with it to see what this means:

```
>>> import h11
>>> conn = h11.Connection(our_role=h11.CLIENT)
>>>
```

At first glance, this is going to look odd. We created a "Connection," but there is none of the usual network-related flavor to it. No host name. No port number. What exactly is this connected to? As it turns out, it's not "connected" to anything. Instead, it is an abstract representation of an HTTP/1.1 connection. Let's make a request on the connection:

```
>>> request = h11.Request(method='GET', target='/',
headers=[('Host', 'httpbin.org')])
>>> bytes_to_send = conn.send(request)
>>> bytes_to_send
b'GET / HTTP/1.1\r\nhost: httpbin.org\r\n\r\n'
>>>
```

Notice that the `send()` method of the connection returned a byte string? These are the bytes that need to be sent someplace. To do that, you are on your own. For example, you could create a socket and do it manually:

```
>>> from socket import *
>>> sock = socket(AF_INET, SOCK_STREAM)
>>> sock.connect(('httpbin.org', 80))
>>> sock.sendall(bytes_to_send)
>>>  # Send end of message to mark the end of the request
>>> sock.sendall(conn.send(h11.EndOfMessage()))
>>>
```

Upon sending the request over a socket, the server will respond. Let's read a tiny fragment of the response:

```
>>> data = sock.recv(10)    # Read 10 bytes
>>> data
b'HTTP/1.1 2'
>>>
```

This is just a fragment of what's being sent to us, but let's feed it into the HTTP/1.1 connection object and see how it responds:

```
>>> conn.receive_data(data)
>>> conn.next_event()
NEED_DATA
>>>
```

The `receive_data()` call feeds incoming data to the connection. After you do that, the `next_event()` call will tell you more about what has been received. In this case, the `NEED_DATA` response means that incomplete data has been received (the connection has not received a full response). You have to read more data. Let's do that.

```
>>> data = sock.recv(1000)
>>> conn.receive_data(data)
>>> conn.next_event()
Response(status_code=200, headers=[(b'server', b'nginx'),
(b'date', b'Tue, 28 Jun 2016 15:31:50 GMT'),
(b'content-type', b'text/html; charset=utf-8'),
(b'content-length', b'12150'), (b'connection', b'keep-alive'),
(b'access-control-allow-origin', b'*'),
(b'access-control-allow-credentials', b'true')], http
_version=b'1.1')
>>>
```

Aha! Now we see the response and some headers. There is a very important but subtle aspect to all of this. When using the protocol library, you just feed it byte fragments without worrying about low-level details (e.g., splitting into lines, worrying about the header size, etc.). It just works. If more data is needed, the library lets you know with the `NEED_DATA` event.

After getting the basic response, you can move on to reading data. To do that, call `conn.next_event()` again.

```
>>> conn.next_event()
Data(data=bytearray(b"<!DOCTYPE html>\n<html>\n<head>\n..")
>>>
```

This is a block of data found in the last read of the underlying sock. You can continue to call `conn.receive_data()` and `conn.next_event()` to process the entire stream.

```
>>> conn.next_event()
NEED_DATA
>>> data = sock.recv(100000)     # Get more data
>>> conn.receive_data(data)
>>> conn.next_event()
Data(data=bytearray(b'p h3+pre {margin-top:5px}\n .mp img ...')
>>> conn.next_event()
EndOfMessage(headers=[])
>>>
```

The `EndOfMessage` event means that the end of the received data has been reached. At this point, you're done. You've made a request and completely read the response.

The really critical part of this example is that the HTTP/1.1 protocol handling is completely decoupled from the underlying socket. Yes, bytes are sent and received on the socket, but none of that is mixed up with protocol handling. This means that you can make the I/O as crazy as you want to make it.

A More Advanced Example: HTTP/2

The hyper-h2 library implements a similar idea as the previous example, but for the HTTP/2 protocol [3]. HTTP/2 is a much more complicated beast, but here is an example of what it looks like to drive it with h2. This program makes a request to `https://http2bin.org` and prints out the events that are returned.

```
from socket import *
import ssl

# Establish a socket connection with TLS
sock = socket(AF_INET, SOCK_STREAM)
sock.connect(('http2bin.org', 443))
ssl_context = ssl.create_default_context()
sock = ssl_context.wrap_socket(sock, server_hostname=
       'http2bin.org')

# Create HTTP/2 connection
import h2.connection
import h2.events

conn = h2.connection.H2Connection(client_side=True)
conn.initiate_connection()
sock.sendall(conn.data_to_send())

# Get the response to the initial connection request
data = sock.recv(10000)
events = conn.receive_data(data)
for evt in events:
   print(evt)

# Send out a request
request_headers = [
   (':method', 'GET'),
   (':authority', 'http2bin.org'),
   (':scheme', 'https'),
   (':path', '/')
]
conn.send_headers(1, request_headers, end_stream=True)
sock.sendall(conn.data_to_send())

# Read all responses until stream is ended
done = False
while not done:
   data = sock.recv(100000)
   events = conn.receive_data(data)
   for evt in events:
      print(evt)
      if isinstance(evt, h2.events.StreamEnded):
         done = True
```

If you run this, you should get output like this:

```
<RemoteSettingsChanged changed_settings:{3:
ChangedSetting(setting=3,
original_value=None, new_value=100),
4: ChangedSetting(setting=4, original_value=65535,
new_value=16777216)}>
<SettingsAcknowledged changed_settings:{}>
<ResponseReceived stream_id:1, headers:[(':status', '200'),
('server', 'h2o/1.7.0'),
('date', 'Tue, 28 Jun 2016 16:12:48 GMT'), ('content-type',
'text/html; charset=utf-8'),
('access-control-allow-origin', '*'),
('access-control-allow-credentials', 'true'),
('x-clacks-overhead', 'GNU Terry Pratchett'),
('content-length', '11729')]>
<DataReceived stream_id:1, flow_controlled_length:11729,
data:3c21444f43545950452068746d6c3e0a3c68746d>
<StreamEnded stream_id:1>
```

This probably looks fairly low-level, but the key part of it is that all of the protocol handling and the underlying I/O are decoupled. There is a TLS socket on which low-level `sendall()` and `recv()` operations are performed. However, all interpretation of the data is performed by the `H2Connection` instance. I don't have to worry about the precise nature of those details.

Putting It All Together

Looking at the previous two examples, you might wonder who these libraries are for? Yes, they handle low-level protocol details, but they provide none of the convenience of higher-level libraries. In the big picture, the main beneficiaries are the authors of I/O libraries, such as myself. To see how this might work, here is bare-bones implementation of a requests-like clone using coroutines and Curio:

```
# example.py

from urllib.parse import urlparse
import socket
import curio
import h11

class Response(object):
   def __init__(self, url, status_code, headers, conn, sock):
      self.url = url
      self.status_code = status_code
      self.headers = headers
      self._conn = conn
      self._sock = sock
```

```
    # Asynchronous iteration for a streaming response
    async def __aiter__(self):
        return self

    async def __anext__(self):
        while True:
            evt = self._conn.next_event()
            if evt == h11.NEED_DATA:
                data = await self._sock.recv(100000)
                self._conn.receive_data(data)
            else:
                break

        if isinstance(evt, h11.Data):
            return evt.data
        elif isinstance(evt, h11.EndOfMessage):
            raise StopAsyncIteration
        else:
            raise RuntimeError('Bad response %r' % evt)

    # Asynchronous property for getting all content
    @property
    async def content(self):
        if not hasattr(self, '_content'):
            chunks = []
            async for data in self:
                chunks.append(data)
            self._content = b''.join(chunks)
        return self._content

async def get(url, params=None):
    url_parts = urlparse(url)
    if ':' in url_parts.netloc:
        host, _, port = url_parts.netloc.partition(':')
        port = int(port)
    else:
        host = url_parts.netloc
        port = socket.getservbyname(url_parts.scheme)

    # Establish a socket connection
    use_ssl = (url_parts.scheme == 'https')
    sock = await curio.open_connection(host,
        port,
        ssl=use_ssl,
        server_hostname=host if use_ssl else None)

    # Make a HTTP/1.1 protocol connection
    conn = h11.Connection(our_role=h11.CLIENT)
    request = h11.Request(method='GET',
                          target=url_parts.path,
                          headers=[('Host', host)])
    bytes_to_send = conn.send(request) + conn.send(h11.
EndOfMessage())
    await sock.sendall(bytes_to_send)

    # Read the response
    while True:
        evt = conn.next_event()
        if evt == h11.NEED_DATA:
            data = await sock.recv(100000)
            conn.receive_data(data)
        elif isinstance(evt, h11.Response):
            break
        else:
            raise RuntimeError('Unexpected %r' % evt)

    resp = Response(url, evt.status_code, dict(evt.headers),
conn, sock)
    return resp
```

Using this code, here is an example that makes an HTTP request and retrieves the result using coroutines:

```
import example
import curio

async def main():
    r = await example.get('http://httpbin.org/')
    print(r.status)
    print(r.headers)
    print(await r.content)

curio.run(main())
```

Or, if you want to stream the response back in chunks, you can do this:

```
async def main():
    r = await example.get('http://httpbin.org/bytes:100000')
    with open('out.bin', 'wb') as f:
        async for chunk in r:
            print('Writing', len(chunk))
            f.write(chunk)

curio.run(main())
```

Fleshing out this code to something more worthy of production would obviously require more work. However, it didn't take a lot of code to make a bare-bones implementation. Nor was it necessary to worry too much about underlying mechanics of the HTTP/1.1 protocol. That's pretty neat.

The Future

If it catches on, the whole idea of breaking out network protocols into libraries decoupled from I/O would be an interesting direction for Python. There are certain obvious benefits such as eliminating the need for every I/O library to implement its own protocol handling. It also makes experimental work in I/O handling much more feasible as it is no longer necessary to implement all of that code from scratch.

Although Cory Benfield's work on HTTP/2 handling may be the most visible, one can't help wonder whether a similar approach to other protocols might be useful. For example, isolating the protocols used for database engines (MySQL, Postgres), Redis, ZeroMQ, DNS, FTP, and other network-related technologies might produce interesting results if it enabled those protocols to be used in different kinds of I/O libraries. It seems that there might be opportunities here.

References

[1] Curio project: https://curio.readthedocs.io.

[2] C. Benfield, "Building Protocol Libraries the Right Way," PyCon 2016: https://www.youtube.com/watch?v=7cC3_jGwl_U.

[3] Hyper-h2 Project: http://python-hyper.org/projects/h2/.

[4] H11 Project: https://github.com/njsmith/h11.

[5] Requests Project: https://docs.python-requests.org.

[6] Urllib3 Project: https://urllib3.readthedocs.io.

Writing for *;login:*

We are looking for people with personal experience and expertise who want to share their knowledge by writing. USENIX supports many conferences and workshops, and articles about topics related to any of these subject areas (system administration, programming, SRE, file systems, storage, networking, distributed systems, operating systems, and security) are welcome. We will also publish opinion articles that are relevant to the computer sciences research community, as well as the system adminstrator and SRE communities.

Writing is not easy for most of us. Having your writing rejected, for any reason, is no fun at all. The way to get your articles published in *;login:*, with the least effort on your part and on the part of the staff of *;login:*, is to submit a proposal to login@usenix.org.

PROPOSALS

In the world of publishing, writing a proposal is nothing new. If you plan on writing a book, you need to write one chapter, a proposed table of contents, and the proposal itself and send the package to a book publisher. Writing the entire book first is asking for rejection, unless you are a well-known, popular writer.

;login: proposals are not like paper submission abstracts. We are not asking you to write a draft of the article as the proposal, but instead to describe the article you wish to write. There are some elements that you will want to include in any proposal:

- What's the topic of the article?
- What type of article is it (case study, tutorial, editorial, article based on published paper, etc.)?
- Who is the intended audience (syadmins, programmers, security wonks, network admins, etc.)?
- Why does this article need to be read?
- What, if any, non-text elements (illustrations, code, diagrams, etc.) will be included?
- What is the approximate length of the article?

Start out by answering each of those six questions. In answering the question about length, the limit for articles is about 3,000 words, and we avoid publishing articles longer than six pages. We suggest that you try to keep your article between two and five pages, as this matches the attention span of many people.

The answer to the question about why the article needs to be read is the place to wax enthusiastic. We do not want marketing, but your most eloquent explanation of why this article is important to the readership of *;login:*, which is also the membership of USENIX.

UNACCEPTABLE ARTICLES

;login: will not publish certain articles. These include but are not limited to:

- Previously published articles. A piece that has appeared on your own Web server but has not been posted to USENET or slashdot is not considered to have been published.
- Marketing pieces of any type. We don't accept articles about products. "Marketing" does not include being enthusiastic about a new tool or software that you can download for free, and you are encouraged to write case studies of hardware or software that you helped install and configure, as long as you are not affiliated with or paid by the company you are writing about.
- Personal attacks

FORMAT

The initial reading of your article will be done by people using UNIX systems. Later phases involve Macs, but please send us text/plain formatted documents for the proposal. Send proposals to login@usenix.org.

The final version can be text/plain, text/html, text/markdown, LaTex, or Microsoft Word/Libre Office. Illustrations should be EPS if possible. Raster formats (TIFF, PNG, or JPG) are also acceptable, and should be a minimum of 1,200 pixels wide.

DEADLINES

For our publishing deadlines, including the time you can expect to be asked to read proofs of your article, see the online schedule at www.usenix.org/publications/login/publication_schedule.

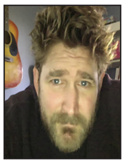
COPYRIGHT

You own the copyright to your work and grant USENIX first publication rights. USENIX owns the copyright on the collection that is each issue of *;login:*. You have control over who may reprint your text; financial negotiations are a private matter between you and any reprinter.

usenix
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

iVoyeur
Pager Trauma Statistics Daemon, PTSD

DAVE JOSEPHSEN



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

I began writing this article late in the third and final day at Monitorama, the single-track monitoring conference. In fact, as I wrote these first few paragraphs, a man named Joey Parsons from Airbnb was talking about the health and well-being of Airbnb's on-call workers. In fact, "people matter" was something of a theme this year—a pretty drastic change for a normally tools and techniques focused crowd. It is certainly true, for whatever reason, that this Monitorama took a heavy turn toward introspection about measurement culture and especially on-call pain.

Over the last several years, a few speakers have talked about what can only be described as post-traumatic reactions to other people's ringtones, wherein they'll react in visceral, lizard-brain terror to the sound of someone else's phone, or even just ambient noise that sounds similar to their own phone's notification tone. This resonates very strongly with me, and I suspect that if you're reading this, it might resonate with you as well.

In 2012 I took a much needed break from Operations to travel the conference circuit, speaking and writing in the name of developer advocacy, outreach, and evangelism. By the time I finally quit Ops to find a job without an on-call component, I'd been on-call for seven years. Yep, every day and every night, as the (usually) sole technical lead for one startup or another.

I think I still underestimate the stress I put on myself and my family during this nearly decade-long on-call stint. I spent three consecutive Thanksgiving dinners alone at a datacenter in what became a running joke at my last startup: the "Thanksgiving Day Curse." The last time this happened in 2012, when my phone went off shortly after I sat down for Thanksgiving dinner, my exasperated wife dumped the contents of my plate into a large zip-lock bag and handed it to me on the way out the door.

Talks on "alert fatigue" have popped up now and again. I've even given one myself, but these normally focus on tools and techniques to reduce and filter false-positives. It seems to me that we rarely talk about the more intangible "people problems" that on-call work can introduce into our lives, or attempt to quantify the amount of pain on-call participants experience on a weekly basis. There are what can only be described as support groups, like the "I made a huge mistake" BoF Shawn Sterling has thrown for years at LISA, but these have always been tertiary events, rather than main-event fodder.

But because we are engineers, and especially because we are engineers who enjoy the measuring and quantification of things, many of the talks at Monitorama this year went beyond descriptions of the problem and described efforts to actually assess the cognitive and emotional stress that the computational first-responders in their organizations have to put up with. Many shared data, either in the form of graphs or spreadsheets, and a few even shared HR techniques like sleep-tracking, and mandatory three-day weekends for individuals coming off their on-call rotation.

I especially found the numbers fascinating. In Joey Parsons' talk "Monitoring and Health at Airbnb" [1], he shared that they've had some success at Airbnb making on-call a voluntary endeavor that is spread among engineers across every internal discipline. Airbnb's 50(!) on-call volunteers take three-day shifts and participate in weekly sysop meetings to keep everyone up to date with respect to chronic, ongoing production issues. The top on-call volunteer for the week before Monitorama fielded 48 production issues.

One of the most important things enabling teams like Airbnb to quantify their on-call stress is the use of third-party alert processing systems like PagerDuty and VictorOps. These systems export APIs that expose a wealth of information about every incident. Who was on-call when it happened, to whom it was assigned, the number of actual notifications that each incident spawned and on and on.

As I write this, I'm also in the process of transitioning back to Ops, so as you can imagine I'm certainly interested in seeing how our own on-call endeavors line up. We have, of course, no hope of finding 50 on-call volunteers in a company of < 20 people, but the ebb and flow of our production issues is a line I'd like to see. To that end, on the plane ride home I spent some time playing with the PagerDuty API, and landed with a rough first-pass at a Go program that can interrogate the PagerDuty API for total incidents, per-user pager notifications, and per-user acknowledgments, and forward these as counter increments to StatsD for visualization.

For the moment I'm calling it PTSD (Pager Trauma Statistics Daemon), and you can find it at http://github.com/djosephsen/ptsd. My hope is that it'll make it trivial for anyone using PagerDuty to get some stats about their on-call volume. It dumps these stats to STDOUT and a locally running StatsD instance.

Using Go

Increasingly, I find myself reaching for Go over something like Bash or Python to write API glue code like this. And I'd like to share the reasons with you, since I feel like it's rare to see real talk about Go in a context other than Web services and hardcore systems tooling. Here, I want to write some longish-term maintainable glue code to tie together a REST API and a service listening on a network socket.

Traditionally, Go has not been my first choice for parsing JSON responses from Web APIs. The built-in net/http library is fantastic for running queries, but the built-in JSON library, encoding/json, is a complete pain when it comes to working with large, complex, or unknown structures. In these cases I almost always find myself decoding into a hash-map (map[string]interface), and then manually type-casting my way to the real values. This process is error-prone, painful, and fragile, and you wind up with line upon line of incomprehensible nested expressions like:

```
'foo['people'].(map[string]interface)['dave'].(string)'
```

By comparison Bash and Python have lovely JSON parsing capabilities, but I've never really cared for their HTTP clients, which are sufficiently complicated that I've never really been able to commit their syntax to memory. Python, especially, is somewhat of a mess in this regard, with `urllib` vs. `requests` vs. `httplib`. To be clear, I'm a pretty down to earth practitioner who doesn't have anything in particular against any of these libraries, but it's basically a nightmare to have to go back and forth between them depending on the project/people I'm working on/with. I've seriously considered picking one and just severing ties with all of the people who use the others.

Recently, however, I discovered Anton Holmquist's "Jason" library [3], and I have to say it's made Go my…well, go-to language for working with JSON Web-APIs. Here's the pattern for making a GET request to an API and parsing the JSON response into something we can work with:

```
client := &http.Client{}
req, _ := http.NewRequest("GET", url, nil)
req.Header.Add("Authorization", authToken)
resp,err := client.Do(req)
body,err := jq.NewObjectFromReader(resp.Body)
```

First, we make a new `http.Client` object. This is unnecessary unless you need to do advanced things to the request like add headers. You can make simple requests directly with `http.Post()`. In this case, I add a header to submit my auth token. Once the request is built, we send it with `client.Do()`, and then we parse the response body directly into a JSON object with `NewObjectFromReader`, which we've imported from `antonholmquist/jason`.

Once the response is in a `json.Object`, a vast and powerful assortment of functions enable us to read out top-level or nested values straight into strings, ints, and etc. without any typecasting on our part. For example, to parse out a top-level `type` attribute into a Go-native string variable we would:

```
type_of_thingie := body.GetString("type")
```

There are even functions that return '[]*jason.Object,' which makes it completely trivial to iterate through arrays nested inside the JSON.

```
logs, _ := body.GetObjectArray("log_entries")
for _,log := range logs{
 fmt.Printf("log type :: %s", log.getString( "type"))
}
```

Needless to say, if I ever meet Anton Holmquist and he needs anything, like a ride home or a free beer or a dude with a truck to help him move, I'll happily oblige him. He's made my life much easier.

The second reason I chose Go was because I was going to want this to be a somewhat extensible project, by which I mean, I wanted to make it as easy as possible for other people to extend it without having to refactor. That meant implementing both the "collector" piece (e.g., PagerDuty) as well as the "outputter" piece (e.g., StatsD) as modules from the...well, get-go.

There are two reasons I find Go pretty nice for writing modular code. The first of these is the Package system. Suffice to say, within the same Go "Package" you can pretty much drop files in the package directory and Go will just use them. I've written a lot of top-level framework code that accepts user-contributed extensions in languages like Python (see, for example, Graphios), and it's just never as easy as it should be. I'm always having to manually include things at the very least, or more likely, fiddle around with ugly stuff like `__init__py` and `sys.path.insert`.

With PTSD, if you want to implement `VictorOps`, you just make a `victorops.go` file in the package dir and implement the Collector interface. That's pretty much it. The `go build` tooling will pick your file right up along with the rest of it and do the right thing. Literally zero messing around with meta-import magic.

The second reason I like writing modular code is the type system. Types are just so easy to create in Go, and interfaces are, IMO, a much more straightforward way of modeling logic in a systems-programming context than Java-style classes. Go interfaces let you reason about the *functionality* that's common between objects rather than the attributes or data-structures they have in common. In this context, writing a new PTSD Collector means creating a type that implements three simple functions (methods, whatever).

`Enabled` takes no arguments and returns a Boolean indicating whether or not PTSD should enable your collector. This allows us to import all available collector add-ons and just switch on the ones we want. The PagerDuty collector's `Enabled` function returns true if you've set a `PDTOKEN` environment variable containing your PagerDuty token.

`Run` takes an `int` and returns nothing. This `int` is PTSD's polling interval (60 minutes by default). The PagerDuty collector uses this interval to decide how far back it should query the PD API for records of incidents.

Finally, `Name` returns the name of the Collector as a string. This is really just used for debugging (e.g., "enabled collector: <name>").

That's it. Just implement those three functions, and the rest of PTSD knows how to deal with your code, and your code can pretty much do anything it wants vis-à-vis actually collecting metrics and sending them to the Outputters via the global `increment` function. When I'm dealing with class-based systems like Python that use object inheritance, I somewhat ironically wind up creating much more formal and therefore usually less flexible object definitions.

The final reason I chose Go was to make it easy for other people to actually use PTSD in real life. This really boils down to the fact that I get a compiled binary as output so I don't have to worry about moving the toolchain around with the executable. This makes things mind-numbingly simple when I want to embed this in a Docker container or toss it on a Heroku Dyno. Really, I know that this thing I just made is going to just run pretty much wherever I throw it, without any `bundler`, `pip`, `npm`, or any other kind of toolchain dependency hell, and by extension I know that nobody else who wants to use this will have to experience any of that either.

By the time you read this, PTSD should be stable and in use at Librato. Despite the latent scarring, I'm really looking forward to returning to Ops work again, and I hope I've inspired you to think about quantifying the on-call stress in your organization, and possibly giving Go a whirl if you haven't already.

Take it easy.

References:

[1] Monitorama, Joey Parsons talk: https://www.youtube.com/watch?v=1SlljMU9V5k&feature=youtu.be&t=20704).

[2] PTSD (Pager Trauma Statistics Daemon): http://github.com/djosephsen/ptsd.

[3] Anton Holmquist's "Jason" library: https://github.com/antonholmquist/jason.

Practical Perl Tools

Seek Wise Consul

DAVID N. BLANK-EDELMAN

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

As we build infrastructure that is more and more fluid, the idea of "service discovery" becomes more and more important. Once upon a time, service discovery was trivial. If you wanted to figure out where a particular service resided so you could tell some other thing in your environment how to talk to it, you could walk over to a machine in one of your racks of equipment, read the labels, and point at it. It likely had a fixed IP address in addition to a fixed physical address.

Those halcyon days (which I'm not sure even I am pining for) are almost gone. With the onset of one or any of a list of factors like virtualization, containerization and "cloudy" provisioning, components in a system and the people running them have a much more difficult time locating each other on the fly. There are a number of different good approaches/tools for this problem. Today we are going to look at the open source package Consul from HashiCorp (https://www.consul.io/—you've probably heard of them because they are the makers of Vagrant) and how to interact with it via Perl. If this topic is popular, I'm happy to look at some of the other choices in a future column.

Basic Consul Concepts

The heart of most of these solutions is some sort of distributed/highly available key-value store that provides easy methods for your other infrastructure components to query/interact with it. "Distributed/highly available" in this case means that the package makes it easy to run multiple servers that replicate data between themselves so that if one goes down, your infrastructure continues to hum along. This requires handling all of the gnarly details around this sort of setup (what to do when one goes down and comes back up again with stale data, how to handle network partitions, where are writes handled in the system, deciding on the fly which instance should be "master" and which should be replicas, etc.). The "key-value store" part of the first sentence is not much more complicated than the key-value concept of a Perl hash, so we should feel right at home when we get to working with that portion of Consul.

In practice what this means is that you run a number of Consul servers and Consul agents. The servers are, well, servers. The agents run on or with every service you wish to make discoverable. Their job is to report in to the servers. They do a little bit more than just register their respective service (which you can do with the Perl modules we're going to be discussing). They can perform health checks on your service and register/deregister it from Consul as appropriate when it becomes operational or sick. They also provide query forwarding so Consul queries can be made of both the servers or the agents (who will automatically forward to an appropriate server). These queries can either be via HTTP, or, to make things really easy, via DNS. If you haven't seen this sort of pure magic before, it entails just having the thing that wants to find that service make a DNS query for the right host name (as in {servicename}.service.consul). Super spiffy.

(Not So) Secret Agent Man

As much as I want to dive directly into the Perl part of all of this, I think it will help if we first start out interacting with the system using the built-in agent functionality before we bring in Perl as an external actor. And while I'm making caveats, I'm not really going to demonstrate any of the functionality around health checks or clustering because it basically just works as the doc suggests and is almost orthogonal to the later discussions around Perl. I'm also not going to discuss installation issues—they are covered in the excellent doc at https://www.consul.io/.

So let's look at perhaps the simplest example of using Consul with a single agent (that will also act as a server). This example comes only slightly modified from the Getting Started documentation. There are two ways to tell a Consul agent about a service that it should advertise: through a config file or via the API. Let's do it both ways.

For a config file, we just need to drop in place a tiny JSON file that looks like this:

```
{
  "service": {
    "name": "webserver"
    "port": 80
  }
}
```

and start up the agent in "dev" mode:

```
consul agent -dev -config-dir ./config.d
```

The agent spins up, loads the config, decides it should become a lead server, and is ready to answer requests (I'm leaving all of the output from the somewhat chatty dev mode in place just because I think the election stuff looks cool):

```
==> Starting Consul agent...
==> Starting Consul agent RPC...
==> Consul agent running!
         Node name: 'dNb-MBP.local'
        Datacenter: 'dc1'
            Server: true (bootstrap: false)
       Client Addr: 127.0.0.1 (HTTP: 8500, HTTPS: -1,
                                    DNS: 8600, RPC: 8400)
      Cluster Addr: 172.27.4.103 (LAN: 8301, WAN: 8302)
    Gossip encrypt: false, RPC-TLS: false, TLS-Incoming: false
             Atlas: <disabled>

==> Log data will now stream in as it occurs:

    2016/06/29 17:32:29 [INFO] raft: Node at 172.27.4.103:8300
 [Follower] entering Follower state
    2016/06/29 17:32:29 [INFO] serf: EventMemberJoin:
dNb-MBP.local 172.27.4.103
    2016/06/29 17:32:29 [INFO] serf: EventMemberJoin:
dNb-MBP.local.dc1 172.27.4.103
    2016/06/29 17:32:29 [INFO] consul: adding WAN server
dNb-MBP.local.dc1 (Addr: 172.27.4.103:8300) (DC: dc1)
    2016/06/29 17:32:29 [INFO] consul: adding LAN server
dNb-MBP.local (Addr: 172.27.4.103:8300) (DC: dc1)
    2016/06/29 17:32:29 [ERR] agent: failed to sync remote
state: No cluster leader
    2016/06/29 17:32:31 [WARN] raft: Heartbeat timeout
reached, starting election
    2016/06/29 17:32:31 [INFO] raft: Node at
172.27.4.103:8300 [Candidate] entering Candidate state
    2016/06/29 17:32:31 [DEBUG] raft: Votes needed: 1
    2016/06/29 17:32:31 [DEBUG] raft: Vote granted from
172.27.4.103:8300. Tally: 1
    2016/06/29 17:32:31 [INFO] raft: Election won. Tally: 1
    2016/06/29 17:32:31 [INFO] raft: Node at 172.27.4.103:8300
[Leader] entering Leader state
    2016/06/29 17:32:31 [INFO] raft: Disabling
EnableSingleNode (bootstrap)
    2016/06/29 17:32:31 [INFO] consul: cluster leadership
acquired    2016/06/29 17:32:31 [DEBUG] raft: Node
172.27.4.103:8300 updated peer set (2): [172.27.4.103:8300]

    2016/06/29 17:32:31 [DEBUG] consul: reset tombstone
GC to index 2
    2016/06/29 17:32:31 [INFO] consul: New leader elected:
dNb-MBP.local
    2016/06/29 17:32:31 [INFO] consul: member 'dNb-MBP.local'
joined, marking health alive
    2016/06/29 17:32:32 [INFO] agent: Synced service 'consul'
    2016/06/29 17:32:32 [INFO] agent: Synced service
'webserver'
```

We can then query it either via DNS:

```
$ dig @127.0.0.1 -p 8600 webserver.service.consul

; <<>> DiG 9.8.3-P1 <<>> @127.0.0.1 -p 8600 webserver.service.
consul
; (1 server found)
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 36403
;; flags: qr aa rd; QUERY: 1, ANSWER: 1, AUTHORITY: 0,
ADDITIONAL: 0
;; WARNING: recursion requested but not available

;; QUESTION SECTION:
;webserver.service.consul.  IN   A

;; ANSWER SECTION:
webserver.service.consul. 0 IN   A  172.27.4.103
```

or via a call to the API:

```
$ curl -s http://localhost:8500/v1/catalog/service/
webserver|jq -M
[
  {
    "Node": "dNb-MBP.local",
    "Address": "172.27.4.103",
    "ServiceID": "webserver",
    "ServiceName": "webserver",
    "ServiceAddress": "",
    "ServicePort": 80,
    "ServiceEnableTagOverride": false,
    "CreateIndex": 5,
    "ModifyIndex": 5
  }
]
```

We can also register a service via the API:

```
$ curl -X POST -d @newservice.json http://localhost:8500/v1/
agent/service/register --header "Content-Type:application/json"
```

This does an HTTP POST of the JSON file with this content:

```
{
  "ID": "webconsole",
  "Name": "webconsole",
  "Port": 8080
}
```

And now we can see it in a query (output excerpted):

```
$ dig @127.0.0.1 -p 8600 webconsole.service.consul
' ;; ANSWER SECTION:
webconsole.service.consul. 0  IN   A 172.27.4.103
```

Key-Value Time

In addition to just being able to register and query service records, Consul also offers the more general key-value store functionality à la etcd or zookeeper. HTTP API calls can be made to store/retrieve a value under a certain key. This is useful if you want to also store some configuration info in Consul. A component can come up, check in with Consul, and get not only pointers to the services it might need but also its configuration values. Consul's key-value support is more sophisticated than what Perl's hashes offer, so perhaps the analogy earlier was a little simplistic. In addition to the usual GET/SET operations, it offers transactions (multiple operations at once that either succeed or fail all together), locks, test-before-set and watch for changes functionality.

Simple work with keys just involves PUT or GET operations to the /v1/kv/<key> endpoint. We could do this via curl commands (similar to exactly what we did before), but better yet, let's use this as an excuse to get into the land of Perl.

Perl Meet Consul

One of the purposes of showing all of these command lines is to demonstrate the ease of interacting with the system. Translating these operations directly to an equivalent generic Perl module would be easy:

- curl calls easily become HTTP::Tiny (or whatever the HTTP module of choice is for you) calls.
- DNS queries are easily done via Net::DNS. The only tricky thing here would be to make sure you specify either the port option to Net::DNS::Resolver->new() or be sure to set the port via the port() method to the port that Consul is using.

We've used this stuff time and time again in the past, so instead let's take a quick look at the more customized modules for Consul. The two I'm aware of are "Consul" and "Consul::Simple." The one thing Consul::Simple has that I think is kind of neat (and possible to easily implement while using the other module) is the ability to set a prefix for key-value operations. As the doc says, this essentially gives you "namespaces," meaning you could have keys named "namespace/thing" so you could use different namespaces for different kinds of keys (e.g., "dev/something," "prod/something," etc.). Unfortunately, Consul::Simple hasn't been touched in a couple of years, so I'm going to focus on the module just called "Consul."

You'll be pleased and I suspect unsurprised to know that using this module looks just like the previous command line operations only simpler. Here's how we might set and retrieve key/values:

```
use Consul;

# talk to localhost by default
my $consul = Consul->kv;

# set some values
my $status;
$status = $consul->put( 'lisa2016' => 'boston' );
die "1st put failed" unless defined $status;
$consul->put( 'sreconEU2016' => 'dublin' );
die "2nd put failed" unless defined $status;

# retrieve them
my $response = $consul->get('lisa2016');
print 'LISA is in ' . $response->value . " in 2016\n";
$response = $consul->get('sreconEU2016');
print 'SREconEU is in ' . $response->value . " in 2016\n";

# let's try the namespaces idea
$status = $consul->put( 'conferences/2017/lisa2017'
   => 'san francisco' );
$status = $consul->put( 'conferences/2017/srecon2017'
    => 'san francisco' );
```

```
# returns all of the matching keys
my $pairs = $consul->get_all('conferences/2017');

foreach my $pair (@$pairs) {
   print $pair->key, "\n";
}
```

I'm hoping that the code above is fairly self-explanatory. We no longer have to worry about HTTP endpoints; the module is taking care of all that for us. The other endpoints in the API are also equally available should we want to register or retrieve services, or receive or change internal Consul configuration (for example, around clustering).

I'd encourage you to play with Consul and all it can do. Take care, and I'll see you next time.

Shutting Down Applications Cleanly

KELSEY HIGHTOWER

Kelsey Hightower has worn every hat possible throughout his career in tech, and enjoys leadership roles focused on making things happen and shipping software. Kelsey is a strong open source advocate focused on building simple tools that make people smile. When he is not slinging Go code, you can catch him giving technical workshops covering everything from programming to system administration and distributed systems.
kelsey.hightower@gmail.com

Developers around the world are building new applications at blazing speeds, and with the growing adoption of microservices and continuous delivery, people are shipping applications faster than ever. Most developers focus on getting applications out the door and deployed to production, which can mean neglecting critical parts of an application's life cycle—the shutdown process.

It's easy to understand why the shutdown process is often neglected. The main goal is to deploy applications and keep them running around the clock, not shutting them down. However, the reality is that modern applications shut down as often as they start up thanks to continuous delivery and better deployment tools. As a developer, it's your job to ensure applications are robust and not only start up correctly but also shut down cleanly. Clean shutdowns are the key to zero-downtime deployments and keeping your production systems in a consistent state. Applications that exit abruptly can wreak havoc on system resources caused by leaving network connections open and partial writes to file systems.

Just Give Me the Signal and I'll Do the Rest

On UNIX systems, applications are sent an asynchronous signal to indicate when an application should terminate. The most common signals sent to initiate the shutdown process are SIGINT, SIGTERM, and SIGKILL. SIGINT is sent when the user wants to interrupt a process. For example, say you're running a ping process in the foreground like this:

```
$ ping usenix.org
PING usenix.org (50.56.53.173): 56 data bytes
64 bytes from 50.56.53.173: icmp_seq=0 ttl=51 time=66.741 ms
64 bytes from 50.56.53.173: icmp_seq=1 ttl=51 time=65.001 ms
^C
--- usenix.org ping statistics ---
2 packets transmitted, 2 packets received, 0.0% packet loss
round-trip min/avg/max/stddev = 65.001/65.871/66.741/0.870 ms
```

Notice how pressing the Ctrl-C key combination causes the ping process to terminate. Also notice how the ping process was able to print a few additional lines before terminating. This works because the ping process caught the SIGINT signal, which made it stop pinging usenix.org and print the results of the ping session.

The SIGTERM signal is sent to a process in order to request that the process terminates. Both the SIGTERM and SIGINT signals can be caught by a running process and cause it to shut down; this is not a hard requirement as a process is free to ignore these signals. The SIGKILL signal is sent to a process to cause it to terminate immediately, and unlike the SIGTERM and SIGINT signals, your process is not given an opportunity to clean up and will be killed—yeah, SIGKILL is brutal. The SIGKILL signal is normally sent after sending a SIGTERM signal and waiting some amount of time before considering the process hung. This is what we want to avoid, so we must make sure the cleanup process happens as quickly as possible.

Don’t Forget to Clean Up

Once a termination signal is received it’s the application’s responsibility to clean up any open network connections, flush pending writes, or complete outstanding client requests before terminating. Let’s write a simple Web application that demonstrates how clean shutdowns work in practice. The Go standard library provides everything required to catch UNIX signals, but we’ll use a third-party library called manners to simplify the handling of incoming HTTP requests during the shutdown process.

Review the following code sample and save it to a file named main.go.

```
package main
import (
    "log"
    "net/http"
    "os"
    "os/signal"
    "syscall"
    "github.com/braintree/manners"
)

func main() {
    log.Println("Starting up...")

    http.HandleFunc("/", func(w http.ResponseWriter, r
               *http.Request) {
        log.Printf("%s %s - %s", r.Method, r.URL.Path,
               r.UserAgent())
        w.Write([]byte("Hello USENIX!\n"))
    })

    // Catch signals in the background using
    // an anonymous go routine.
    go func() {
        signalChan := make(chan os.Signal, 1)
        signal.Notify(signalChan, syscall.SIGINT,
               syscall.SIGTERM)

        // This blocks until this process receives a SIGINT
        // or SIGTERM signal.
        <-signalChan
        log.Println("Shutting down...")

        // stop accepting new requests and begin shutting down.
        manners.Close()
    }()

    // The call to ListenAndServe() blocks until an error is
    // returned or the manners.Close() function is called.
    err := manners.ListenAndServe(":8080",
               http.DefaultServeMux)
    if err != nil {
        log.Fatal(err)
    }
    log.Println("Shutdown complete.")
}
```

The source code can also be cloned from https://github.com/kelseyhightower/hello-usenix:

```
$ git clone https://github.com/kelseyhightower/hello-usenix.git
$ cd hello-usenix
```

Next, fetch the manners package, which provides a wrapper for Go’s standard HTTP server and ensures all active HTTP requests have completed before the HTTP server shuts down.

```
$ go get github.com/braintree/manners
```

Finally, build the hello-usenix server using the go build command:

```
$ go build -o hello-usenix main.go
```

At this point you should have a hello-usenix binary in the current directory that can be used to examine a clean shutdown process in action. The remainder of this tutorial will require three separate terminals where commands can be executed.

Terminal #1

Start the hello-usenix server in the foreground:

```
$ ./hello-usenix
2016/07/03 15:37:30 Starting up...
```

Terminal #2

Use curl to make HTTP requests to the hello-usenix service:

```
$ while true; do curl http://127.0.0.1:8080; sleep 1; done
Hello USENIX!
Hello USENIX!
…
```

Terminal #3

Grab the process ID (PID) of the running hello-usenix process using the ps command:

```
$ ps -u `whoami` | grep hello-usenix
55211 ttys003    0:00.03 ./hello-usenix
```

Use the kill command to send the SIGTERM signal to the hello-usenix process using the correct process ID:

```
$ kill 55211
```

By default the kill command sends the SIGTERM signal to the running process specified by the given process ID (PID). Once the SIGTERM signal reaches the hello-usenix process, the shutdown process will begin by rejecting new HTTP requests, which will cause the curl command running in Terminal #2 to return connection-refused errors:

Terminal #2

```
$ while true; do curl http://127.0.0.1:8080; sleep .2; done
Hello USENIX!
Hello USENIX!
...
curl: (7) Failed to connect to 127.0.0.1 port 8080: Connection
refused
```

Although handling signals and performing a clean shutdown will give your servers the opportunity to tidy up, it won't prevent clients from continuing to send your services requests. In the case of the hello-usenix application, a front-end load balancer can be used to detect when an instance of hello-usenix is no longer accepting HTTP requests and route incoming requests to another instance.

Once the the final HTTP request has completed, the hello-usenix server will print the "Shutdown complete" log message and terminate:

Terminal #1

```
$ ./hello-usenix
2016/07/03 20:42:06 Starting up...
2016/07/03 20:43:03 GET / - curl/7.43.0
2016/07/03 20:43:04 GET / - curl/7.43.0
...
2016/07/03 20:44:24 Shutting down...
2016/07/03 20:44:24 Shutdown complete.
```

That's what a clean shutdown looks like. Catch a few signals and clean up before terminating. It looks so easy, right? Thanks to manners and the Go standard library, it is.

Conclusion

As a developer it's important to think about the complete application life cycle, and that includes the shutdown process. All applications should perform clean shutdowns and tidy up before terminating. Applications that get this right are much easier to manage and reduce the need for complex deployment scripts that attempt to protect clients from servers which accept requests that will never be processed or to clean up database connections left behind by misbehaving applications.

For Good Measure
Paradigm

DAN GEER

Dan Geer is the CISO for In-Q-Tel and a security researcher with a quantitative bent. He has a long history with the USENIX Association, including officer positions, program committees, etc. dan@geer.org

Whether cybersecurity is, or could become, a science is no easy question, and any seemingly easy answer is simplistic. We need a science of security, that much is sure. Yes, we are making, and have made, significant advances in technique, but there is no doubt that something more generative needs to come onto the scene. Consider, via Figure 1, six major advances in technique and their useful lifetime [1].

But what would a science of security be? There is much to think about in revisiting T. S. Kuhn's 1962 landmark work, *The Structure of Scientific Revolutions*. Kuhn begins and ends with what is a circular idea, that a scientific community is defined by what beliefs practitioners share, and what beliefs practitioners share defines what community they are in. This is, in fact, instructive as no science begins in mature form, but rather any new science will begin in much more modest circumstances where, early on, consensus is not even a concept. As such, part of becoming a mature science is the development of a broad consensus about the core concerns of that branch of knowledge. Kuhn's word for the collections of exemplars of good science was "paradigm," a word whose meaning today is all but entirely Kuhn's, even among those who've never read a word he wrote.

But what is a "paradigm" and why do we want one? As Kuhn puts it, "[Paradigms] are the source of the methods, the problem-field, and standards of solution accepted by any mature scientific community at any given time." Kuhn's book and the two-decade-long back and forth between Kuhn and philosophers notwithstanding, the simplest version is that a paradigm is all the things that a scientist can assume that his or her colleagues will congenially understand about their common work without explicitly explaining them or arguing them from first principles again and again.

Again quoting Kuhn, "Men whose research is based on shared paradigms are committed to the same rules and standards for scientific practice. That commitment and the apparent consensus it produces are prerequisites for normal science, i.e., for the genesis and continuation of a particular research tradition.... Acquisition of a paradigm and of the more esoteric type of research it permits is a sign of maturity in the development of any given scientific field." Competing schools of thought are always present before a mature science first appears. As Kuhn said, "What is surprising, and perhaps also unique in its degree to the fields we call science, is that such initial divergences should ever largely disappear. For they do disappear to a very considerable extent and then apparently once and for all." Perhaps it is thus possible to say that topics of study that never coalesce their competing schools are either fated never to be sciences or are in some state of arrested development that may someday be cured. Those that can and will, but have not yet done so, are what Kuhn called pre-paradigmatic, meaning not yet a science. The appearance of a paradigm that all can accept is when the transition to being a science occurs, or, as Kuhn put it, "Except with the advantage of hindsight, it is hard to find another criterion that so clearly proclaims a field a science."

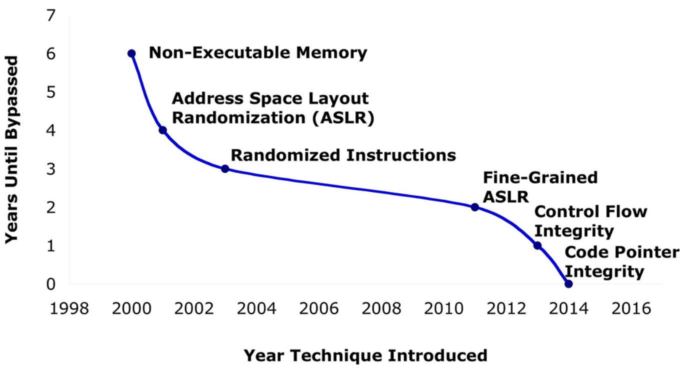


Figure 1: Major advances in technique and their useful lifetime

For Kuhn, the appearance of a paradigm transforms those who merely study first into a discipline and then into a profession. One can even say, and Kuhn does, that the paradigm itself is the last result of the science in question that can be appreciated by the lay audience—after that, all progress is in journal articles not readable by non-specialists, to the extent that "[t]he scientist who writes [for the lay reader] is more likely to find his professional reputation impaired than advanced" for having done so.

Now as everyone here knows, from time to time a science may undergo a revolution, which in Kuhn's terms is precisely the laying down of one paradigm in preference for another. The title of his book is to be understood as precisely that, that scientific revolutions share aspects of structure that we can now describe as there have been enough of them in the last 400 years to discern that structure. If you consider physics to be the paragon of a hard science, then the transition from Newtonian mechanics to Einsteinian relativity demonstrates exactly the point Kuhn was making, that there comes a moment when research has reached a kind of impasse where the nature of what now look to be puzzles needing further study cannot be profitably investigated within the paradigm that now holds.

Kuhn referred to these impasses as the appearance of an anomaly, one that the existing paradigm cannot evaluate by way of further research consistent with the paradigm then in place. His review of past revolutions centered in each case on the appearance of irreconcilable anomalies that made a given field ripe for revolution. That roasting metals caused them to gain weight thus indicating that they had absorbed some fraction of the air around them, a fraction that could be exhausted, led to the idea that air might not be the one and only gas but rather a combination of gases. Perhaps more significantly to the very idea of revolution is that even though Lavoisier had discovered oxygen, others in the field, notably Priestly, never accepted the existence of oxygen and held to the phlogiston theory to the end of their careers. I say "more significantly" as the trite version of "What is a scientific revolution?" is that it is a time when newcomers to the field adopt the new paradigm while those already in the field slowly die off. It is a generational change.

Kuhn's idea of crisis is the dual of his idea of paradigm. Where a science's paradigm suggests puzzles that further research will solve, in a crisis this is no longer so. Yet the occasional crisis is itself necessary for advancement as any paradigm whose theories completely explain all observable fact ceases to be science and becomes engineering. In other words, a crisis is not the end of research but the substitution of a new paradigm for an old and a new set of research puzzles awaiting solution.

Just as "a scientific theory is declared invalid only if an alternate candidate is available to take its place," to reject one paradigm without simultaneously substituting another is "to reject science itself." In short, when an anomaly appears, there are only three resolutions available: (1) solve the problem, (2) leave the problem for future scientists, or (3) use the crisis to force a new paradigm on the field.

When there is a shift of paradigm, that is to say a scientific revolution, it may serve to redirect a field so completely that some parts of it fall away entirely, the separation of astronomy from astrology or the separation of chemistry from physics being two examples. As Kuhn put it, the choice between paradigms is a choice between incompatible modes of community life (and, yes, he does mean "scientific community" in an altogether social sense). "Since no two paradigms leave all the same problems unsolved, paradigm debates always involve the question: Which problems is it more significant to have solved?"

One of the first questions we might ask is whether cybersecurity is a science or, if not, whether it ever will be. I am one of several expert reviewers for the National Security Agency's annual "Science of Security" competition and award [2]. Quoting its rationale, "The competition was established to recognize the current security paper that best reflects the conduct of good science in the work described. [Science of Security] is a broad enterprise, involving both theoretical and empirical work. While there can only be one best paper, any one paper cannot span that full breadth. Nonetheless, the field is broad and work in all facets is encouraged and needed. The common denominator across the variety of approaches is solid methodology and effective communication, so those aspects of the papers [are] strong factors in our decision" [3].

Papers are nominated for consideration, and I encourage you to do so, but I am also here to report that among the reviewers our views of what constitutes a, or the, Science of Security vary rather a lot. Some of us would prioritize purpose, agreeing with Charles Darwin that "all observation must be for or against some view if it is to be of any service" [4]. Some of us view aspects of methodology as paramount, especially reproducibility and the clarity of communication on which it depends. Some of us are ever on the lookout for what a physicist would call a unifying field theory. Some of us insist on the classic process of hypothesis generation followed by designed experiments. We vary, and I take that to be a vote of sorts on whether cybersecurity is yet a science.

Whether cybersecurity is yet a science is a hard question. Let's consider candidate paradigms of cybersecurity; if they exist and have turned over from time to time then, yes, cybersecurity is now a science. Take one of the most basic tools we employ, that of authentication. Authentication is the solution to the puzzle of identity establishment, a puzzle that derived from the paradigm of perimeter control.

The paradigm of perimeter control has been in an evident crisis for some time now. The crisis is not merely because the definition of perimeter may have been poorly applied in practice, but because some combination of always-on and universal addressability collectively make the paradigm of a defensible perimeter less and less a paradigm where research is itself likely to patch up the mess and retain the core and guiding paradigm of perimeter control. The parallel with Ptolemaic astronomy is pretty fair. On the one hand, every improvement in observational accuracy made the motions of the planets more complicated to describe with epicycles upon epicycles. On the other hand, our hand, the threat to systems from always-on universal addressability has become too rich to be just a new set of puzzles solely within the paradigm of perimeter control—the defensible perimeter began to have its own version of epicycles within epicycles by a shrinking of what a perimeter could, or should, control [5].

A second crisis for the paradigm of perimeter control is upon us now as exemplified with a commercial example. Table 1 counts cores in the Qualcomm Snapdragon 801.

Central CPU	4
Adreno 330 GPU	4
Video out	1
Hexagon QDSP	3
Modem	2–4
Bluetooth	1
USB controller	1
GPS	1
Wifi	1–2
Charging	?
Power	?
Display	?

Table 1: Identifiable cores in Qualcomm Snapdragon 801

That is somewhere between 18 and 21 cores. In the vocabulary of the Internet of Things, is that one "thing" or the better part of two dozen "things"? Is the perimeter to be defended the physical artifact in the user's pocket or is it the execution space of each of those cores? The compound annual growth rate of deployed "things" approximates 35%—meaning a 27-month doubling time. That's a lot of new perimeter.

If the paradigm of perimeter control is no longer producing puzzles that can be solved by further scientific research, then what? Noting, as Kuhn does, that "to reject one paradigm without simultaneously substituting another is to reject science itself," what might be a substitution? If everything we are or do is unique if examined closely enough, then the idea of authentication as verifying an assertion like "My name is Dan" can easily morph into an observable like "Sensors say that this is Dan." In other words, our paradigm of an authentication transaction before any other perimeter-piercing transaction is itself showing its age.

The paradigm that is the obvious alternative to perimeter control, and thus authentication as a gating function, is accountability based on one single unspoofable identity per person. If I am right—that real-soon-now identity is simply an observable that needs no assertions—then that single identity which the individual has but does not need to prove may be fast upon us. The National Strategy for Trusted Identities in Cyberspace is not worded in that way although that is how I read it. That means there is a crisis in privacy research, too. Privacy's paradigm has long been, "Privacy is the power to selectively reveal oneself to the world" [6], but all uses of virtual reality come with total surveillance.

Nevertheless, if being part of the modern world in no more robust way than appearing unmasked on a public street is the same as submitting to unitary identification observable at a distance by things you never heard of, then that means either submission or withdrawal for the individual.

Note that Kuhn never said that a switch to a new paradigm would be delightful or comforting, he merely said that it would better explain the way the world works while suggesting new puzzles for scientists who share that paradigm to pursue. Authentication transactions as a prodrome to authorization transactions in service to a paradigm of perimeter control may soon be behind us, including in the peer-to-peer world. If, in fact, being authenticated as yourself is unavoidable, then there is no proving that this is the Dan for whom there is a book entry allowing him into some robot-protected building, but rather an accountability regime based on whether that Dan did or did not enter a building for which he might later be penalized. His crime would not be masquerading as some identity other than his own so as to get in, but rather that of he was observed to have gone in even though he was forbidden to do so.

What I am suggesting as the crisis around the paradigm of selective revelation is that, as with metadata, there is so much redundancy in what is observable that prohibiting one or another form of collection has no meaningful effect whatsoever on those agencies, whether intelligence or advertising, who would build a model of you from metadata alone. As but one example, with current technology I can read the unique radio signature of your beating heart at five meters. As with anything that has an electromagnetic output, the only technological question is the quality of the antenna. If I can take your picture on the public street without your permission or notice, why can't I record your heart? Or your iris? Or your gait? Or the difference in temperature between your face and your hands? That list is long and getting longer. It is a crisis for which the paradigm of selective revelation can scarce put up puzzles fast enough, and scientific solving of those puzzles can, at best, trail the curve.

The crisis is simply that what heretofore we have known as confidentiality is becoming quaint and irrelevant. Perhaps science will have to reposition confidentiality within some new paradigm that prioritizes integrity, not confidentiality. Perhaps a world in which data can and will be collected irrespective of selective permission granting is a world in which the data had better be right. If more and more intelligent actors are to be out there doing our implicit bidding long after we've forgotten their configuration interface, then data integrity had better be as absolute as we can make it, and that is then where the research puzzles will have to be found.

Perhaps I have it wrong, perhaps the topmost paradigm of the science of security is simply that of defense. Perhaps the rise of sentient opponents makes a paradigm of defense unarguable, as evidenced by rafts of paradigmatically generated puzzles of the sort of how can this or that be hardened or otherwise defended, up to and including DARPA's Grand Challenge [7].

If defense is and has been our paradigm, then that, too, is in crisis. That is in no way a failure; paradigms only change due to the success that the one paradigm has in motivating science to explore the world thoroughly enough to discover anomalies that cannot be made to fit within the paradigm that caused them to be discovered in the first place. The outgrowth of the paradigm of defense has been guidance that has allowed us, including non-scientist practitioners, to get better and better. We have discovered and then deployed better tools, we have come to understand causal chains and thus have achieved better understood practices and work with better colleagues. That's the plus side, and it is one terrific plus side. But if I am interested in the ratio of skill to challenge, then, as far as I can estimate, we are expanding the society-wide attack surface faster than our science of security is expanding our collection of tools, practices, and colleagues. The paradigm of defense is in crisis.

One embodiment of the paradigm of defense has been the movement to build security in. The successes of that movement are precisely of the sort I mentioned before when I said that we have discovered and then deployed better tools. But to remind you of the truism in Adi Shamir's 2002 Turing Award lecture, "Cryptography is typically bypassed, not penetrated." I would argue that this is true of all aspects of the cybersecurity mechanism, including those delivered by building security in; it is the possibility of bypass that ultimately matters. Our sentient opponents know that, too, and their investments in automating the discovery of methods of bypass are in a hell of a horse race with both building security in and in-static analysis of code bodies, new or old. Look (again) at Figure 1.

Kuhn takes some pains to say why it is that a paradigm shift requires a crisis, that "to an extent unparalleled in other fields, [scientists] have undergone similar educations and professional initiations." One here must ask the central question of this essay by mirroring Kuhn: are the paradigms of cybersecurity in enough of a crisis that resolution of the crisis requires a change of paradigm? The answer is by no means obvious, although to my eye there are several crises now in play. If the crises sufficient to require a reformulation of the paradigm or paradigms of cybersecurity, then a scientific revolution is upon us, what Kuhn calls "a reconstruction of group commitments." As he points out, a crisis requiring such a reconstruction might not even be in cybersecurity itself, but instead might be due to discoveries in some other field or venue, just as discoveries in physics engendered a crisis in chemistry once upon a time.

That, then, is the question before us, complicated by the changing nature of what scientists of security are studying both with respect to rapid technological change and the presence of sentient opponents, leavened, of course, with the societal demands fast upon us largely independent of what we know or say. I think I see paradigms here that are in undeniable crisis. I can, of course, be entirely wrong, and we may still be working our way up to being a science, still coalescing schools of thought into the kind of paradigm that will define us as scientists.

A fuller treatment of this topic is available at http://geer.tinho.net/geer.nsf.6i15.txt.

References

[1] Chart data courtesy of Hamed Okhravi, MIT.

[2] NSA, Science of Security: www.nsa.gov/what-we-do/research/science-of-security/index.shtml.

[3] Best Scientific Cybersecurity Paper winner: https://www.nsa.gov/news-features/press-room/press-releases/2014/best-scientific-cybersecurity-paper-competition.shtml.

[4] Letter to Henry Fawcett, 1863, as quoted in E. J. Huth and T. J. Murray, eds., *Medicine in Quotations: Views of Health and Disease through the Ages* (American College of Physicians, 2006), p. 169.

[5] IT Conversations, "The Shrinking Security Perimeter," recorded March 1, 2004: audio mirror at: geer.tinho.net/Dan_Geer_-_The_Shrinking_Security_Perimeter.mp3.

[6] E. Hughes, "A Cypherpunk's Manifesto," March 9, 1993: http://www.activism.net/cypherpunk/manifesto.html.

[7] "Cyber Grand Challenge": https://cgc.darpa.mil/.

/dev/random
Distributed Illogic

ROBERT G. FERRELL

Robert G. Ferrell is an award winning author of humor, fantasy, and science fiction, most recently *The Tol Chronicles* (www.thetolchronicles.com).
rgferrell@gmail.com

As a teen I bought my first car (a '69 Chevy Impala Custom, in the trunk of which you could park most of today's models with room to spare) using money I'd saved from various jobs after school and on weekends. The engine was a small block 350 without any fancy electronics, emissions control devices apart from an exhaust manifold, or fuel injection mumbo-jumbo. I knew how it worked, how to do routine maintenance and simple repairs, and where everything was in the engine compartment. I could replace/gap spark plugs, change the oil, filters, and distributor cap, adjust the timing, play with the carb mixture, and so on. It was a straightforward, reliable vehicle, even if it did lack certain optional luxuries like functional motor mounts.

When I open the hood of my 2001 Trans Am (yeah, I still drive that wonderful dinosaur, when I drive at all), even after 15 years of ownership I'm frankly at a loss to understand any more than half of what I see in there. I'm lucky if I can find the oil dipstick, to be brutally honest. So much has changed in the world of automotive technology since 1974. I plug in my diagnostic computer readout thingamajig whenever I get a warning light on the dashboard, but I don't have any idea what the messages it displays are talking about most of the time. I just shrug and hit "erase all."

Technology in virtually every area has advanced significantly, of course. Take distributed computing, for example. It wasn't that long ago that the suggestion that bits and pieces of not only our data but the very software and hardware that process it would be scattered hither and yon across the Internet like propaganda leaflets dropped from a vintage South Korean Piper Cub would have been met with ridicule or at least eye rolls and head-shaking.

I figure if we're going to continue down the distributed everything road, we may as well take it to the next level and distribute the electricity that runs these machines while we're at it. So let's say I need 30 amps at 110 volts to run a server rack that's processing on the cloud. In the old system, I would simply plug into a UPS that then was fed by a circuit from the local electrical utility (in most cases). How quaint. In my sleek, modern power supply engineering paradigm, nodes all over the planet advertise the number of electrons they have sitting around unused at the moment and send them wherever they're needed on request.

At any given moment, then, you might be powering your cloud server with juice from Montevideo, Edmonton, Aberdeen, Zagreb, Taipei, Jakarta, and Perth. I'm not certain, but that could require some conversion from European or Asian volts to North American volts. I think electrons might travel on the other side of the wire in most of those countries, too, but we can work with that.

As an adjunct to the distributed computing trend, I propose we stop calling it the "Internet" and adopt "Omninet." It has a more inclusive ring, don't you think? With the term Omninet you don't need to make cumbersome distinctions like "the Internet of Things" because

Omninet pretty much covers all that ground by default. It also has a sort of Orwellian *the government is watching you* feel to it that should prove popular amongst certain elements of today's (justifiably) paranoid society.

Maybe it is also time we consider taking our distributed physical architecture to its logical extreme. Why stop at the board level when you can drill on down to individual components? We can assemble the necessary circuits on the fly from a database of hundreds of thousands or even millions of resistors, capacitors, diodes, integrated circuits, and so on available worldwide using the new generation of just-in-time hardware compilers I recently made up. That way the only piece of processing hardware you actually need on premises is the compiler itself. Everything else can be recruited from the Omninet in real time. Maybe we could even figure out a way to assemble the compiler on the fly. Closed loops are so entertaining.

Having a ten thousand-mile-long electrical bus might seem a little ponderous, but think of the local thermal advantages, not to mention the savings from not needing to buy equipment that becomes obsolete before you can get it installed and configured. Heck, I see significant advantages even for home users, especially gamers. One of the reasons I finally gave up on PCs and went to console gaming exclusively some years ago is that I got tired of having to upgrade my video card for every new astronomical-polygon count game release. Sixty bucks for the game and another three hundred for the hardware to run it puts a real dentaroo in the ol' household budget, know what I mean? I have a cabinet drawer that could supply the nucleus of a decent graphics card museum.

If the game itself could actually specify its minimum hardware requirements for running and recruit the necessary components from the Omninet, that would be just super. Sure, a few itty-bitty latency issues might crop up at first, but I'm certain they'll be overcome. After all, we can stream 4K cat videos to corners of the planet where indoor plumbing is considered a novelty. All a gamer would need is a monitor, an Omninet connection, and whatever interface was necessary to make the distributed components work together.

Of course, some might argue, why bother with on-premises hardware at all? Just use a sort of distributed Steam-like system where each player forks a new instantiation of both the game and platform. Simplifies multiplayer quite a bit. We might take clipping to new heights, as well: instead of merely hiding the parts of a rendered scene not currently visible to the player, don't even *write the code* until the predictive engine determines it will be required soon. Who needs human programmers these days, anyway? Am I right? Step away from the pitchfork, meatbag.

If I've said all this before, I'm not surprised. Three days is about the maximum I can go without repeating myself at dinner table conversation with my wife; I've been writing this column now for over ten years. You're lucky I don't just select random paragraphs from previous columns and string them together. I called it "/dev/random" for a *reason,* you know.

Um, actually, that's not a bad idea. Reusable code is all the rage, after all. Maybe I can put every paragraph I've ever written into a database and let you mash them up yourselves: sort of a literary mix tape. This sort of betrayal only hurts if you let it.

Register Now!

OSDI 16

12th USENIX Symposium on Operating Systems Design and Implementation

Sponsored by USENIX in cooperation with ACM SIGOPS

November 2–4, 2016 • Savannah, GA

Join us in Savannah, GA, November 2–4, 2016, for the **12th USENIX Symposium on Operating Systems Design and Implementation (OSDI ’16)**. The Symposium brings together professionals from academic and industrial backgrounds in what has become a premier forum for discussing the design, implementation, and implications of systems software.

Co-located with OSDI ’16 on Tuesday, November 1:

• **INFLOW ’16:** 4th Workshop on Interactions of NVM/Flash with Operating Systems and Workloads

The full program and registration are now available.
Register by Friday, October 7, and save!

www.usenix.org/osdi16

usenix THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

BOOKS

Book Reviews

MARK LAMOURINE

Introducing Go
Caleb Doxsey
O'Reilly Media, 2016, 112 pages
ISBN: 978-1-491-94195-9

I didn't look twice when I saw this title come up as a "people also bought" on Amazon. A lot of my work recently has been developing or using applications written in Go(lang), and I've been looking for good books on Go for more than a year. For the longest time there were no professionally published books on Go, and the only resources were blogs, the official godoc Web site (https://godoc.org), and a couple of Creative Commons texts written by developers mostly found on the Golang Web site (https://golang.org/doc/). (Don't misunderstand me. These are great resources, but I like books, made of paper, in my hand.) Recently this has changed dramatically: there are half a dozen books on learning Go in my local bookstore. I'm really glad about this and have started picking up books to read.

When I first open a box of books, I take each one out and leaf through it before setting it on the pile of "to be read" books. I stopped when I got to *Introducing Go* and looked more carefully. It turns out that I'd read and reviewed the same book in 2014. Well, not quite the same book.

Caleb Doxsey was one of the first authors to publish a book on learning Go. It was first available as a Web site in collaboration with Google, then a PDF entitled "An Introduction to Programming in Go" made available by Doxsey and Google under a CC3 license in 2012. It's still there if you want to read it (http://www.golang-book.com/books/intro). For a time the book was also available as a paperback, but the conversion from e-text to paper was a mixed success.

Introducing Go is, at its core, Doxsey's original corpus but published by O'Reilly. It's still a slim volume and the contents and flow are largely lifted from the earlier work, but that's not what's important. The text has been updated and reformatted, and this is a significant improvement over the self-published version from 2012. It's not that Doxsey didn't do a good job, but O'Reilly really enforces good editing and layout: the code examples stand out and are much easier to read in the standard typeset style; the paragraphing and tables are clearer; and although the book is physically smaller, what's there is what's important and it's clear.

There is a small set of books that I call the "slim classics." I'm thinking of *The C Programming Language* (Kernighan and Ritchie), *Unix Programming Environment* (Kernighan and Pike), and a couple of others like them. They're not always the best all-around guides or references, but they distill the essence of a topic in a way that thicker tomes sometimes lose in their quest for authoritative completeness. In these days of fast-moving language development, I don't know if there's room for slim classics, but I think *Introducing Go* could be a contender.

Go in Action
William Kennedy, with Brian Ketelsen and Erik St Martin
Manning Publications Company, 2016, 241 pages
ISBN: 978-1-61729-178-1

As just mentioned, I've been on a bit of a Go(lang) book kick recently. *Go in Action* is one I looked forward to. In general I like Manning's style, and their editorial choices tend to walk the fine line between traditional dusty references and frothy Dummies-style tutorials.

The authors used the MEAP (Manning Early Access Program) process for writing and pre-release editing and commentary, in which early subscribers get to see the chapters raw as they are submitted, and the authors and editors get commentary during the writing process. In other words, they use the Internet to apply the "many eyes" principle to writing.

After reading a number of texts on Go, I see a kind of standard narrative path emerging: "Hello, world," general development and build environment, pulling packages for inclusion, then into the language constructs themselves, finally ending with a chapter on testing. This is a perfectly reasonable path to take, but after reading a number of them, I'm finding it harder to see what distinguishes one book from another.

Go in Action does have a number of aspects that set it apart. The first I noted above: Manning has developed a very clean typography and layout style, which makes following the commentary and code very easy. Line numbering each code block makes following references easy both for the authors and for the reader. Using a registration table inside the front cover of the book, Manning offers a watermarked copy of every book in PDF, EPUB, and MOBI for every paper copy purchased. Their focus on writing for both paper and electronic documents means that the two forms have parity. I can read the EPUB version and get as good an experience as when reading the hard copy.

Once you get below the general arc of these books, Kennedy et al. do a very good job of showing both how to use Go constructs and what happens when you do. The Go runtime environment is very different from anything that readers coming from scripting languages or Java might be familiar with. Each language construct has a deliberate effect and behavior in the runtime, and unlike modern scripting languages, Go is explicitly designed not to hide the underlying mechanisms. You can stick your hand in the running motor if you want to. Kennedy et al. provide text and diagrams that illustrate these behaviors. This helps new developers avoid (or in my case, recognize after the fact) the pitfalls that can lead to lost fingers.

This is not an introductory book on programming. It's likely to be too much for a reader who isn't already proficient in one or several other languages.

The code examples are available online as are updates to the e-text (assuming you've registered your copy). The authors are available by email or other means for questions or commentary.

If you're coming to Go as a student or professional developer, *Go in Action* would not be a bad introduction. I don't think you'll want to stop there, though.

Docker in Action

Jeff Nickoloff
Manning Publications, 2016, 284 pages
ISBN 978-1-63343-023-5

As with the Go language, there has been a shortage of good books on Docker. There is a relationship between the two. Docker is written in Go, and Go and Docker both have reached a level of maturity and stability where it makes sense to begin writing about them, and *Docker in Action* does a good job.

Often when people try to explain software containers they begin with Docker's shipping container metaphor. Unfortunately, this isn't really an apt metaphor. It's neither insightful or informative when applied to software containers. Then they start trying to define them as "not virtual machines" which is similarly uninformative.

Nickoloff opens *Docker in Action* with one of the best descriptions of software containers that I've seen (though he does at one point tip a hat to shipping containers). Containers are just processes with blinders on, and Nickoloff shows clearly how they relate to the OS, to VMs, and traditional processes.

I really like the progression of the narrative in this book. The author begins with simple containers doing simple jobs locally. All of the examples involve real-world tasks, using containers to replace the traditional applications. Along the way he discusses both the benefits and costs of this. Every case shows the CLI command, which invokes the container, the overt result, and then goes under the covers to show how the result was achieved in the context of containers. When readers follow the text carefully, they will know "what happened" at each of the appropriate layers. This is really important when readers put their understanding to use doing new tasks.

Another thing I really like about the book is Nickoloff's restraint when it comes to building new containers. Dockerfiles and custom images are sexy and interesting looking, but for containers to fulfill their promise, most people should be using off-the-shelf images. Nickoloff manages to get more than half way through the text before discussing container builds. Even then he treats it as a small step, merely extending existing images and moving right on to continuous integration and publishing, the true life cycle of container images. I suspect he knows that there are other texts that go into Dockerfile management in painful detail and that the "best practices" for creating new images for composition are still being developed.

I did learn quite a bit about a more open topic in container management: orchestration. Docker Inc. has been developing in-house tools for composing containers into applications. Others have been doing similar work, but Docker Inc. would really like you to use theirs. This final section introduces Docker's offerings.

Docker Compose is Docker's answer to Kubernetes. It provides a way to create containers that are meant to work in unison to form an application or service. Docker Machine (which was once known as "boot to docker") is a tuned bootable image meant solely to host the Docker runtime. It is an analog of CoreOS or Project Atomic, both minimized bootable images meant to host container runtime environments. Docker Swarm is meant to allow for scaling and distribution of containers across a multi-host environment. Again, this is comparable to Kubernetes or OpenShift.

Each of these tools gets a chapter and a little more at the end of the book. The examples and illustrations are every bit as good here as they are in the chapters covering the more mature elements of Docker. Even though I'm familiar with Kubernetes, I expect I will turn back here the next time I need to build a multi-container application quickly. Once I am more familiar with the Docker tools, I can evaluate whether they will fill the needs of the kinds of large-scale systems that Kubernetes means to build. Nickoloff has made it easy for me to explore a tool set I would otherwise not have considered given my current experience. Make of that what you will.

Of the few texts I've seen on Docker so far, this is the one I would hold out to someone who asked me where to start. Nicely done.

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, system administration articles, tips and techniques, practical columns on such topics as security, Perl, networks and operating systems, and book reviews

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

Discounts on registration fees for all USENIX conferences

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

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

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

USENIX Board of Directors

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

Why USENIX?

by Carolyn Rowland, USENIX Board President

My first LISA conference was in 1994 in Monterey, California. I was a lone sysadmin professionally, and I had never seen that many IT operations people under one roof. There were people who had authored the tools the rest of us used, and it seemed like a gathering of every kind of systems person you could imagine. I mostly kept to myself in the early days. Eventually several of us on my team attended LISA and it turned into a week of collecting ideas and then coming together to bond and talk about what we had heard. I changed jobs, but LISA continued to be a place to go to get exposure to interesting ideas and to hear people from across all industries discuss how they faced all kinds of challenges.

Eventually I realized there was also a great big community of people at the conference. The community was multi-faceted, so finding the groups that interested me meant frequenting Birds-of-a-Feather sessions (BoFs) and other social activities, such as board game night and the reception. I made a bunch of great professional contacts with whom I still talk today. That's when I realized that I didn't need to bring my own community to LISA; there was a whole community waiting for me. Somehow the "hallway track," or community interaction, was just as valuable as the formal conference content.

At some point I volunteered, and there I was, on a LISA program committee. I was reading draft LISA paper submissions with my industry perspective, and it was a tiny peek into the inner workings. I chaired LISA in 2012 and realized that it was rushing by too quickly. I loved trying to create and nurture communities through shared interests, but chairing LISA was a firehose of project planning with no time to try to grok how it all worked before it was time to deliver the conference program to USENIX. I tried to leave behind some breadcrumbs for the chairs that came after me, but my real chance to make a difference came when a USENIX Board member approached me and asked me to consider running for a seat in the upcoming Board election. This would not only mean getting more involved in USENIX from a broader perspective, but it would also give me a chance to slow down and potentially have a bigger impact on LISA, my own community.

Upon joining the Board, I found that there were many communities under the USENIX umbrella and many of them had champions on the Board. This was fascinating as we talked about ways that each of us wanted to help our communities. David Blank-Edelman and I became the LISA co-champions on the Board. We talked to volunteers and attendees. We heard from people who wanted to update the conference. There were so many ideas, most of them excellent. How would we execute the plans? How could we improve on the LISA model in ways that would get the community more excited about training and talks? How could we address longstanding concerns about the complexity of the program or the sheer volume of information we were cramming into a week? How do people learn best and how can we enable people in our community to stay relevant when it seemed like technology was only moving faster all the time? Would we find a way?

The LISA of today looks a bit different from the LISA of 2012, and it feels good to see the USENIX staff, Board, and community working together to make it happen. But it's more than just LISA.

To me, USENIX is that small mom-and-pop shop you love to champion, where the service is tip-top and the community is supportive and accepting. In a world where negativity, bullying, misogyny, and other threats to your being become more mainstream, this isn't a small achievement. USENIX is also a freedom fighter in the battle for open access to information. This is a hard battle to win. Behind the scenes, the staff are often burning the candle at both ends to organize, produce, promote, and publish talks, papers, journals, and *;login:* in order for you to have access to that information as soon as possible as well as free-and-clear. Of course, nothing is free to produce. USENIX is continually looking for ways to ensure that we continue to support open access. USENIX isn't a huge corporation—we are a nonprofit organization—and there isn't a long line of companies champing at the bit to line the open access coffers. So why do it? We do it because it's the right thing to do. Research and knowledge should be available to all.

This means that our events tend toward strong content in talks, panels, tutorials, and workshops without much glitz and flash. It means the staff wear many hats as they work toward the goal. It means the goal keeps changing because the world keeps changing and there are so many thriving communities to serve.

Looking into the machinery, I am captivated by the strength and passion of the USENIX team. These attributes extend beyond the staff team to the Board of Directors, who volunteer their service, and the community members who act as steering committee, program, and organizing committee members; program chairs; speakers; instructors; and researchers, all who share a common goal.

If there is one thing I would still like to do, it would be an expansion of good works. We have been focused on the fiscal health of USENIX and making sure the staff are not overworked as they produce the events that feed our current communities. I would love to look at the makeup of these communities and bring in more students and under-represented folks. As we begin to reach the point of financial sustainability, we can do more in this effort financially, but it's not too soon to start strategizing about it.

There's definitely more work to be done. I look forward to partnering with the USENIX staff, the rest of the Board, and members like you to nurture and grow our communities to be even more inclusive, supportive, and essential to advancing computing research and enhancing our professional lives.

Announcement and Call for Papers www.usenix.org/fast17/cfp

FAST '17: 15th USENIX Conference on File and Storage Technologies

February 27–March 2, 2017 • Santa Clara, CA

Sponsored by USENIX in cooperation with ACM SIGOPS

usenix THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

Important Dates

- Paper submissions due: Tuesday, September 27, 2016, 9:00 p.m. PDT
- Tutorial submissions due: Tuesday, September 27, 2016, 9:00 p.m. PDT
- Notification to authors: Monday, December 12, 2016
- Final paper files due: Tuesday, January 31, 2017

Conference Organizers

Program Co-Chairs

Geoff Kuenning, *Harvey Mudd College*
Carl Waldspurger, *CloudPhysics*

Program Committee

Nitin Agrawal, *Samsung Research*
Irfan Ahmad, *CloudPhysics*
Remzi Arpaci-Dusseau, *University of Wisconsin-Madison*
Mahesh Balakrishnan, *Yale University*
André Brinkmann, *Johannes Gutenberg University Mainz*
Haibo Chen, *Shanghai Jiao Tong University*
Peter Desnoyers, *Northeastern University*
Ashvin Goel, *University of Toronto*
Ajay Gulati, *ZeroStack*
Danny Harnik, *IBM Research - Haifa*
Kimberly Keeton, *Hewlett Packard Labs*
Ricardo Koller, *IBM Research*
Sungjin Lee, *Inha University*
Tudor Marian, *Google*
Marshall Kirk McKusick, *McKusick Consultancy*
Arif Merchant, *Google*
Brad Morrey, *Hewlett Packard Labs*
Sam H. Noh, *UNIST (Ulsan National Institute of Science and Technology)*
Raju Rangaswami, *Florida International University*
Erik Riedel, *EMC*
Jiri Schindler, *Simplivity*
Bianca Schroeder, *University of Toronto*
Philip Shilane, *EMC*
Keith A. Smith, *NetApp*
Swaminathan Sundararaman, *Parallel Machines*
Vasily Tarasov, *IBM Research*
Eno Thereska, *Confluent and Imperial College London*
An-I Andy Wang, *Florida State University*
Hakim Weatherspoon, *Cornell University*
Sage Weil, *Red Hat*
Theodore M. Wong, *Human Longevity, Inc.*
Gala Yadgar, *Technion—Israel Institute of Technology*
Erez Zadok, *Stony Brook University*
Ming Zhao, *Arizona State University*

Steering Committee

Remzi Arpaci-Dusseau, *University of Wisconsin—Madison*
William J. Bolosky, *Microsoft Research*
Angela Demke Brown, *University of Toronto*
Greg Ganger, *Carnegie Mellon University*
Garth Gibson, *Carnegie Mellon University and Panasas, Inc.*
Casey Henderson, *USENIX Association*
Kimberly Keeton, *HP Labs*
Florentina Popovici, *Google*
Erik Riedel, *EMC*
Jiri Schindler, *SimpliVity*
Bianca Schroeder, *University of Toronto*
Margo Seltzer, *Harvard University and Oracle*
Keith A. Smith, *NetApp*
Eno Thereska, *Confluent and Imperial College London*
Ric Wheeler, *Red Hat*
Erez Zadok, *Stony Brook University*
Yuanyuan Zhou, *University of California, San Diego*

Overview

The 15th USENIX Conference on File and Storage Technologies (FAST '17) brings together storage-system researchers and practitioners to explore new directions in the design, implementation, evaluation, and deployment of storage systems. The program committee will interpret "storage systems" broadly; everything from low-level storage devices to information management is of interest. The conference will consist of technical presentations including refereed papers, Work-in-Progress (WiP) reports, poster sessions, and tutorials.

FAST accepts both full-length and short papers. Both types of submissions are reviewed to the same standards and differ primarily in the scope of the ideas expressed. Short papers are limited to half the space of full-length papers. The program committee will not accept a full paper on the condition that it is cut down to fit in the short paper page limit, nor will it invite short papers to be extended to full length. Submissions will be considered only in the category in which they are submitted.

Topics

Topics of interest include but are not limited to:

- Archival storage systems
- Auditing and provenance
- Caching, replication, and consistency
- Cloud storage
- Data deduplication
- Database storage
- Distributed storage (wide-area, grid, peer-to-peer)
- Empirical evaluation of storage systems
- Experience with deployed systems
- File system design
- High-performance file systems
- Key-value and NoSQL storage
- Memory-only storage systems
- Mobile, personal, and home storage
- Parallel I/O and storage systems
- Power-aware storage architectures
- RAID and erasure coding
- Reliability, availability, and disaster tolerance
- Search and data retrieval
- Solid state storage technologies and uses (e.g., flash, byte-addressable NVM)
- Storage management
- Storage networking

- Storage performance and QoS
- Storage security
- The challenges of big data and data sciences

New in 2017! Deployed Systems

In addition to papers that describe original research, FAST '17 also solicits papers that describe large-scale, operational systems. Such papers should address experience with the practical design, implementation, analysis, or deployment of such systems. We encourage submission of papers that disprove or strengthen existing assumptions, deepen the understanding of existing problems, and validate known techniques at scales or in environments in which they were never before used or tested. Deployed-system papers need not present new ideas or results to be accepted, but should offer useful guidance to practitioners.

Authors should indicate on the title page of the paper and in the submission form that they are submitting a deployed-system paper.

Submission Instructions

Please submit full and short paper submissions (no extended abstracts) by 9:00 p.m. PDT on September 27, 2016, in PDF format via the Web submission form on the FAST '17 Web site, www.usenix.org/fast17/cfp. Do not email submissions.

- The complete submission must be no longer than 12 pages for full papers and 6 pages for short papers, excluding references. The program committee will value conciseness, so if an idea can be expressed in fewer pages than the limit, please do so. Supplemental material may be appended to the paper without limit; however the reviewers are not required to read such material or consider it in making their decision. Any material that should be considered to properly judge the paper for acceptance or rejection is not supplemental and will apply to the page limit. Papers should be typeset on U.S. letter-sized pages in two-column format in 10-point Times Roman type on 12-point leading (single-spaced), with the text block being no more than 6.5" wide by 9" deep. Labels, captions, and other text in figures, graphs, and tables must use reasonable font sizes that, as printed, do not require extra magnification to be legible. Because references do not count against the page limit, they should not be set in a smaller font. **Submissions that violate any of these restrictions will not be reviewed.** The limits will be interpreted strictly. No extensions will be given for reformatting.
- Templates and sample first pages (two-column format) for Microsoft Word and LaTeX are available on the USENIX templates page, www.usenix.org/templates-conference-papers.
- **Authors must not be identified in the submissions, either explicitly or by implication.** When it is necessary to cite your own work, cite it as if it were written by a third party. Do not say "reference removed for blind review." Any supplemental material must also be anonymized.
- Simultaneous submission of the same work to multiple venues, submission of previously published work, or plagiarism constitutes dishonesty or fraud. USENIX, like other scientific and technical conferences and journals, prohibits these practices and may take action against authors who have committed them. See the USENIX Conference Submissions Policy at www.usenix.org/conferences/submissions-policy for details.
- If you are uncertain whether your submission meets USENIX's guidelines, please contact the program co-chairs, fast17chairs@usenix.org, or the USENIX office, submissionspolicy@usenix.org.
- Papers accompanied by nondisclosure agreement forms will not be considered.

Short papers present a complete and evaluated idea that does not need 12 pages to be appreciated. Short papers are not workshop papers or work-in-progress papers. The idea in a short paper needs to be formulated concisely and evaluated, and conclusions need to be drawn from it, just like in a full-length paper.

The program committee and external reviewers will judge papers on technical merit, significance, relevance, and presentation. A good research paper will demonstrate that the authors:

- are attacking a significant problem,
- have devised an interesting, compelling solution,
- have demonstrated the practicality and benefits of the solution,
- have drawn appropriate conclusions using sound experimental methods,
- have clearly described what they have done, and
- have clearly articulated the advances beyond previous work.

A good deployed-system paper will demonstrate that the authors:

- are describing an operational system that is of wide interest,
- have addressed the practicality of the system in more than one real-world environment, especially at large scale,
- have clearly explained the implementation of the system,
- have discussed practical problems encountered in production, and
- have carefully evaluated the system with good statistical techniques.

Moreover, program committee members, USENIX, and the reading community generally value a paper more highly if it clearly defines and is accompanied by assets not previously available. These assets may include traces, original data, source code, or tools developed as part of the submitted work.

Blind reviewing of all papers will be done by the program committee, assisted by outside referees when necessary. Each accepted paper will be shepherded through an editorial review process by a member of the program committee.

Authors will be notified of paper acceptance or rejection no later than Monday, December 12, 2016. If your paper is accepted and you need an invitation letter to apply for a visa to attend the conference, please contact conference@usenix.org as soon as possible. (Visa applications can take at least 30 working days to process.) Please identify yourself as a presenter and include your mailing address in your email.

All papers will be available online to registered attendees no earlier than Tuesday, January 31, 2017. If your accepted paper should not be published prior to the event, please notify production@usenix.org. The papers will be available online to everyone beginning on the first day of the main conference, February 28, 2017. Accepted submissions will be treated as confidential prior to publication on the USENIX FAST '17 Web site; rejected submissions will be permanently treated as confidential.

By submitting a paper, you agree that at least one of the authors will attend the conference to present it. If the conference registration fee will pose a hardship for the presenter of the accepted paper, please contact conference@usenix.org.

If you need a bigger testbed for the work that you will submit to FAST '17, see PRObE at www.nmc-probe.org.

Best Paper Awards

Awards will be given for the best paper(s) at the conference. A small, selected set of papers will be forwarded for publication in ACM *Transactions on Storage* (TOS) via a fast-path editorial process. Both full and short papers will be considered.

Test of Time Award

We will award a FAST paper from a conference at least 10 years earlier with the "Test of Time" award in recognition of its lasting impact on the field.

Work-in-Progress Reports and Poster Sessions

The FAST technical sessions will include a slot for short Work-in-Progress (WiP) reports presenting preliminary results and opinion statements. We are particularly interested in presentations of student work and topics that will provoke informative debate. While WiP proposals will be evaluated for appropriateness, they are not peer reviewed in the same sense that papers are. We will also hold poster sessions each evening. WiP submissions will automatically be considered for a poster slot, and authors of all accepted full papers will be asked to present a poster on their paper. Other poster submissions are very welcome. Please see the Call for Posters and WiPs, which will be available soon, for submission information.

Birds-of-a-Feather Sessions

Birds-of-a-Feather sessions (BoFs) are informal gatherings held in the evenings and organized by attendees interested in a particular topic. BoFs may be scheduled in advance by emailing the Conference Department at bofs@usenix.org. BoFs may also be scheduled at the conference.

Tutorial Sessions

Tutorial sessions will be held on February 27, 2017. Please submit tutorial proposals to fasttutorials@usenix.org by 9:00 p.m. PDT on September 27, 2016.

Registration Materials

Complete program and registration information will be available in December 2016 on the conference Web site.

Announcement and Call for Papers www.usenix.org/nsdi17/cfp

NSDI '17: 14th USENIX Symposium on Networked Systems Design and Implementation

March 27–29, 2017 • Boston, MA

Sponsored by USENIX, the Advanced Computing Systems Association

Important Dates

- Paper titles and abstracts due: **Wednesday, September 14, 2016, 3:00 p.m. US PDT**
- Full paper submissions due: **Wednesday, September 21, 2016, 3:00 p.m. US PDT**
- Notification to authors: **Monday, December 5, 2016**
- Final paper files due: **Thursday, February 23, 2017**

Symposium Organizers

Program Co-Chairs

Aditya Akella, *University of Wisconsin–Madison*
Jon Howell, *Google*

Program Committee

Sharad Agarwal, *Microsoft*
Tom Anderson, *University of Washington*
Andrea Arpaci-Dusseau, *University of Wisconsin—Madison*
Anirudh Badam, *Microsoft*
Mahesh Balakrishnan, *Yale University*
Fabian Bustamante, *Northwestern University*
Ranveer Chandra, *Microsoft*
David Choffnes, *Northeastern University*
Romit Roy Choudhury, *University of Illinois at Urbana–Champaign*
Mosharaf Chowdhury, *University of Michigan*
Mike Dahlin, *Google*
Anja Feldmann, *Technische Universität Berlin*
Rodrigo Fonseca, *Brown University*
Nate Foster, *Cornell University*
Deepak Ganesan, *University of Massachusetts Amherst*
Phillipa Gill, Stony *Brook University*
Srikanth Kandula, *Microsoft*
Teemu Koponen, *Styra*
Sanjeev Kumar, *Uber*
Swarun Kumar, *Carnegie Mellon University*
Wyatt Lloyd, *University of Southern California*
Boon Thau Loo, *University of Pennsylvania*
Jacob Lorch, *Microsoft*
Ratul Mahajan, *Microsoft*
Dahlia Malkhi, *VMware*
Dave Maltz, *Microsoft*
Z. Morley Mao, *University of Michigan*
Michael Mitzenmacher, *Harvard University*
Jason Nieh, *Columbia University*
George Porter, *University of California, San Diego*
Luigi Rizzo, *University of Pisa*
Srini Seshan, *Carnegie Mellon University*
Anees Shaikh, *Google*
Ankit Singla, *ETH Zürich*
Robbert van Renesse, *Cornell University*
Geoff Voelker, *University of California, San Diego*
David Wetherall, *Google*
Adam Wierman, *California Institute of Technology*
John Wilkes, *Google*
Minlan Yu, *University of Southern California*
Heather Zheng, *University of California, Santa Barbara*
Lin Zhong, *Rice University*

Steering Committee

Katerina Argyraki, *EPFL*
Paul Barham, *Google*
Nick Feamster, *Georgia Institute of Technology*
Casey Henderson, *USENIX Association*
Arvind Krishnamurthy, *University of Washington*
Jeff Mogul, *Google*
Brian Noble, *University of Michigan*
Timothy Roscoe, *ETH Zürich*
Alex C. Snoeren, *University of California, San Diego*

Overview

NSDI focuses on the design principles, implementation, and practical evaluation of networked and distributed systems. Our goal is to bring together researchers from across the networking and systems community to foster a broad approach to addressing overlapping research challenges.

NSDI provides a high-quality, single-track forum for presenting results and discussing ideas that further the knowledge and understanding of the networked systems community as a whole, continue a significant research dialog, or push the architectural boundaries of network services.

Topics

We solicit papers describing original and previously unpublished research. Specific topics of interest include but are not limited to:

- Highly available and reliable networked systems
- Security and privacy of networked systems
- Distributed storage, caching, and query processing
- Energy-efficient computing in networked systems
- Cloud/multi-tenant systems
- Mobile and embedded/sensor applications and systems
- Wireless networked systems
- Network measurements, workload, and topology characterization systems
- Self-organizing, autonomous, and federated networked systems
- Managing, debugging, and diagnosing problems in networked systems
- Virtualization and resource management for networked systems and clusters
- Systems aspects of networking hardware
- Experience with deployed/operational networked systems

- Communication and computing over big data on a networked system
- Practical aspects of network economics
- An innovative solution for a significant problem involving networked systems

Operational Systems Track

In addition to papers that describe original research, NSDI '17 also solicits papers that describe the design, implementation, analysis, and experience with large-scale, operational systems and networks. We encourage submission of papers that disprove or strengthen existing assumptions, deepen the understanding of existing problems, and validate known techniques at scales or environments in which they were never used or tested before. Such operational papers need not present new ideas or results to be accepted.

Authors should indicate on the title page of the paper and in the submission form that they are submitting to this track.

What to Submit

NSDI '17 is **double-blind,** meaning that authors should make a good-faith effort to anonymize papers. This is new for NSDI in 2017. As an author, you should not identify yourself in the paper either explicitly or by implication (e.g., through the references or acknowledgments). However, only non-destructive anonymization is required. For example, system names may be left un-anonymized, if the system name is important for a reviewer to be able to evaluate the work. For example, a paper on experiences with the design of .NET should not be re-written to be about "an anonymous but widely used commercial distributed systems platform."

Additionally, please take the following steps when preparing your submission:

- Remove authors' names and affiliations from the title page.
- Remove acknowledgment of identifying names and funding sources.
- Use care in naming your files. Source file names, e.g., Joe.Smith.dvi, are often embedded in the final output as readily accessible comments.
- Use care in referring to related work, particularly your own. Do not omit references to provide anonymity, as this leaves the reviewer unable to grasp the context. Instead, a good solution is to reference your past work in the third person, just as you would any other piece of related work.
- If you need to reference another submission at NSDI '17 on a related topic, reference it as follows: "A related paper describes the design and implementation of our compiler [Anonymous 2017]." with the corresponding citation: "[Anonymous 2017] Under submission. Details omitted for double-blind reviewing."
- Work that extends an author's previous workshop paper is welcome, but authors should (a) acknowledge their own previous workshop publications with an anonymous citation and (b) explain the differences between the NSDI submission and the prior workshop paper.
- If you cite anonymous work, you must also send the deanonymized reference(s) to the PC chair in a separate email.
- Blinding is intended to not be a great burden. If blinding your paper seems too burdensome, please contact the program co-chairs and discuss your specific situation.

Submissions must be no longer than 12 pages, including footnotes, figures, and tables. Submissions may include as many additional pages as needed for references and for supplementary material in appendices. The paper should stand alone without the supplementary material, but authors may use this space for content that may be of interest to some readers but is peripheral to the main technical contributions of the paper. Note that members of the program committee are free to not read this material when reviewing the paper.

Submissions must be in two-column format, using 10-point type on 12-point (single-spaced) leading, with a maximum text block of 6.5" wide x 9" deep, with .25" inter-column space, formatted for 8.5" x 11" paper. Papers not meeting these criteria will be rejected without review, and no deadline extensions will be granted for reformatting. Pages should be numbered, and figures and tables should be legible when printed without requiring magnification. Authors may use color in their figures, but the figures should be readable when printed in black and white. All papers must be submitted via the Web submission form linked from the Call for Papers Web site, www.usenix.org/nsdi17/cfp.

Submissions will be judged on originality, significance, interest, clarity, relevance, and correctness.

Policies

Simultaneous submission of the same work to multiple venues, submission of previously published work, or plagiarism constitutes dishonesty or fraud. USENIX, like other scientific and technical conferences and journals, prohibits these practices and may take action against authors who have committed them. See the USENIX Conference Submissions Policy at www.usenix.org/conferences/submissions-policy for details.

Previous publication at a workshop is acceptable as long as the NSDI submission includes substantial new material. See remarks above about how to cite and contrast with a workshop paper.

Authors uncertain whether their submission meets USENIX's guidelines should contact the Program Co-Chairs, nsdi17chairs@usenix.org.

Papers accompanied by nondisclosure agreement forms will not be considered. All submissions will be treated as confidential prior to publication on the USENIX NSDI '17 Web site; rejected submissions will be permanently treated as confidential.

Ethical Considerations

Papers describing experiments with users or user data (e.g., network traffic, passwords, social network information) should follow the basic principles of ethical research, e.g., beneficence (maximizing the benefits to an individual or to society while minimizing harm to the individual), minimal risk (appropriateness of the risk versus benefit ratio), voluntary consent, respect for privacy, and limited deception. When appropriate, authors are encouraged to include a subsection describing these issues. Authors may want to consult the Menlo Report at www.caida.org/publications/papers/2012/menlo_report_actual_formatted/ for further information on ethical principles, or the Allman/Paxson IMC '07 paper at conferences.sigcomm.org/imc/2007/papers/imc76.pdf for guidance on ethical data sharing.

Authors must, as part of the submission process, attest that their work complies with all applicable ethical standards of their home institution(s), including, but not limited to privacy policies and policies on experiments involving humans. Note that submitting research for approval by one's institution's ethics review body is necessary, but not sufficient—in cases where the PC has concerns about the ethics of the work in a submission, the PC will have its own discussion of the ethics of that work. The PC's review process may examine the ethical soundness of the paper just as it examines the technical soundness.

Processes for Accepted Papers

If your paper is accepted and you need an invitation letter to apply for a visa to attend the conference, please contact conference@usenix.org as soon as possible. (Visa applications can take at least 30 working days to process.) Please identify yourself as a presenter and include your mailing address in your email.

Accepted papers may be shepherded through an editorial review process by a member of the Program Committee. Based on initial feedback from the Program Committee, authors of shepherded papers

Continued on next page ➛

will submit an editorial revision of their paper to their Program Committee shepherd. The shepherd will review the paper and give the author additional comments. All authors, shepherded or not, will upload their final file to the submissions system by the camera ready date for the conference Proceedings.

All papers will be available online to registered attendees before the conference. If your accepted paper should not be published prior to the event, please notify production@usenix.org. The papers will be available online to everyone beginning on the first day of the conference.

Best Paper Awards

Awards will be given for the best paper(s) at the conference.

Community Award

To encourage broader code and data sharing within the NSDI community, the conference will also present a "Community Award" for the best paper whose code and/or data set is made publicly available by the final papers deadline, February 23, 2017. Authors who would like their paper to be considered for this award will have the opportunity to tag their paper during the submission process.

usenix
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

USENIX Association
2560 Ninth Street, Suite 215
Berkeley, CA 94710

POSTMASTER
Send Address Changes to *;login:*
2560 Ninth Street, Suite 215
Berkeley, CA 94710

PERIODICALS POSTAGE
PAID
AT BERKELEY, CALIFORNIA
AND ADDITIONAL OFFICES

LISA16

**More Craft.
Less Cruft.**

Dec. 4–9, 2016
BOSTON

Now in its 30th year, LISA is the premier IT operations conference where systems engineers, operations professionals, and academic researchers share real-world knowledge about designing, building, and maintaining the critical systems of our interconnected world.

KEYNOTE SPEAKERS:
- Jane Adams, Data Research and Acceleration, Two Sigma
- Mitchell Hashimoto, Founder, HashiCorp
- John Roese, CTO, EMC

The complete program is now available.

Register by November 10 and save!
usenix.org/lisa16

Sponsored by the USENIX Association