;login:

JUNE 2015 VOL. 40, NO. 3

usenix
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

File Systems and Storage

usenix 40TH ANNIVERSARY

usenix
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

UPCOMING EVENTS

USENIX ATC '15: 2015 USENIX Annual Technical Conference

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

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

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

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

USENIX Security '15: 24th USENIX Security Symposium

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

Co-located with USENIX Security '15:

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

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

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

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

JETS '15: 2015 USENIX Journal of Election Technology and Systems Workshop
(Formerly EVT/WOTE)
August 11, 2015
www.usenix.org/jets15

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

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

LISA15

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

Co-located with LISA15:

UCMS '15: 2015 USENIX Configuration Management Summit
November 9, 2015

URES '15: 2015 USENIX Release Engineering Summit
November 13, 2015
Submissions due September 4, 2015
www.usenix.org/ures15

FAST '16: 14th USENIX Conference on File and Storage Technologies

February 22–25, 2016, Santa Clara, CA, USA
Submissions due September 21, 2015
www.usenix.org/fast16

NSDI '16: 13th USENIX Symposium on Networked Systems Design and Implementation

Co-located with the 2016 Open Networking Summit
March 16–18, 2015, Santa Clara, CA, USA
Paper titles and abstracts due September 17, 2015
www.usenix.org/nsdi16

OSDI '16: 12th USENIX Symposium on Operating Systems Design and Implementation

November 2–4, 2016, Savannah, GA, USA

Stay Connected...

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

;login:

JUNE 2015 VOL. 40, 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 bi-monthly by the USENIX Association, 2560 Ninth Street, Suite 215, Berkeley, CA 94710.

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

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

©2015 USENIX Association
USENIX is a registered trademark of the USENIX Association. Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. USENIX acknowledges all trademarks herein. Where those designations appear in this publication and USENIX is aware of a trademark claim, the designations have been printed in caps or initial caps.

EDITORIAL

Musings

RIK FARROW

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

While watching Brent Welch present his storage technologies tutorial right before FAST '15, I found myself thinking about the topic of the storage hierarchy: registers, various caches (SRAM), memory (DRAM), then the much slower non-volatile storage mediums: flash, disk, and tape. Brent shared the table he used in his talk (Figure 1), which really got me going in this direction. Again.

I realized that I have taken the path of discussing the enormous disparity between processor speed and even DRAM too many times already. Of course, I am not the first to think about this problem. Admiral Grace Hopper used to hand out one-foot-long pieces of wire and explain that each piece represented one nanosecond, the maximum distance light could travel during that brief moment of time. Her point was to make the most of every instruction, as even the speed of light was a real limitation.

Of course, Hopper worked in an era where a single computer filled a room, long before the beginning of integrated circuits. And the wires she handed out, actually about 11.8 inches long, represented the speed of light in a vacuum: in copper, light only travels about half that far in a nanosecond. In the fiber optics I'll soon be addressing, light travels about eight inches in a nanosecond.

Kimberly Keaton of HP Labs also caught my attention. I was congratulating her for winning a Test of Time award (see the FAST '15 conference report in this issue), along with a crew of other HP Labs employees, and Keaton suggested that I take a look at "The Machine." I'd heard of The Machine, but I'd also heard of something that sounded similar in a ParLabs paper six years ago. Keaton pointed out that this new project was more like the FireBox, a project that Krste Asanović spoke about in his keynote during FAST '14 [2].

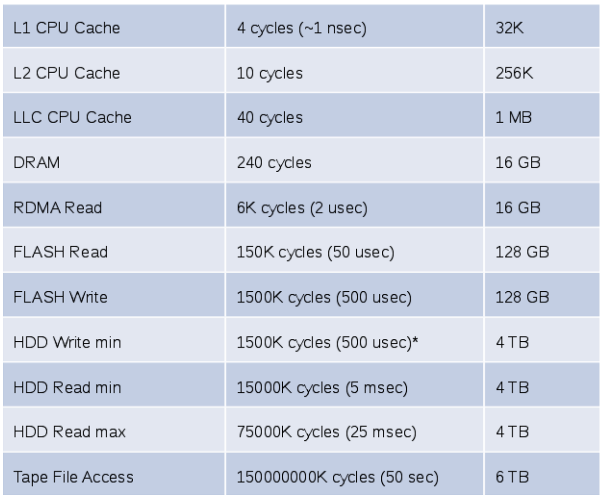
The Machine

So I visited the Web site for The Machine [3], expecting to find the usual marketing there. And there really wasn't much, so I started watching the videos. I skipped the one by HP CTO Martin Fink at first, but he actually does spend more time on the details than the other two videos found on that page (as of March 30, 2015).

But I wanted to know more, so I asked for "someone technical" who could talk about The Machine. HP PR set me up with Kirk Bresniker, an HP fellow and the architect who helped to develop the entire category known as blade servers. Kirk was now working on The Machine, as well as on something called HP Moonshot, which is involved in the project.

First, here's the meat of Fink's talk, at 16:10 into the video: electrons, photons, and ions. Electrons compute, photons communicate, and ions store. Okay, I hope that sounds weird, or at least trite, but there is meat there. There are really three big pieces to The Machine that Fink tells us about, and I want to discuss the last two first.

Like FireBox, The Machine will rely on photonic communications. By using light over fiber, instead of electrons over wires, you can get faster communication. Eliminating wires also helps to deal with design issues because wires have capacitance, which makes communicating over wires slower in more ways than one.



L1 CPU Cache	4 cycles (~1 nsec)	32K
L2 CPU Cache	10 cycles	256K
LLC CPU Cache	40 cycles	1 MB
DRAM	240 cycles	16 GB
RDMA Read	6K cycles (2 usec)	16 GB
FLASH Read	150K cycles (50 usec)	128 GB
FLASH Write	1500K cycles (500 usec)	128 GB
HDD Write min	1500K cycles (500 usec)*	4 TB
HDD Read min	15000K cycles (5 msec)	4 TB
HDD Read max	75000K cycles (25 msec)	4 TB
Tape File Access	150000000K cycles (50 sec)	6 TB

Figure 1: Brent Welch's table showing storage latency and example capacities from his FAST '15 storage technology tutorial

Fink said that The Machine, when working at datacenter scale, can access memory in any rack in just 250 nanoseconds, which, if we ignore issues like switching times, limits the length of fiber that can be used to half of 167 feet, without actually including the reading or writing time for the remote memory involved. There are other problems with photonics, mostly because photonics are still largely research projects. But photonics will be real one day, and I've already heard talks about photonic switching at NSDI.

Next, let's discuss the ions that store. Fink is referencing memristors, a non-volatile memory (NVM) store developed by HP years ago, but still not in production. Memristors are very simple: they use a layer of titanium dioxide at the intersection of nanoscale wires to store bits, which suggests that memristors could reach densities far exceeding that of flash, as well as being byte-addressable in the way that DRAM is—a cache line at a time in practice. Memristors are also expected to be eight times faster than DRAM, making them a lot faster than flash. Looking at Figure 1, that would make memristors close to LLC cache in performance.

Memristors replace both DRAM and long-term storage in The Machine. And I like that idea, of having storage class amounts of NVM with the performance of L3 cache. The balance between memory access times and CPU performance has been very skewed for decades now [4], and this would do a lot to redress that issue. For example, instead of running memcached sharded over many servers, the entire database could fit into a single system with memristor memory and run *faster* than it can run in DRAM. Of course, your SQL database could also fit into NVM as well, presuming NVM becomes as common as multi-terabyte disks are today. Any of today's big data applications would truly benefit from memristors.

If they existed.

The Reality

And there's the rub: neither memristors nor photonics are available today.

I spent a lot of time listening to Kirk Bresniker talking, as he obviously is used to giving presentations to analysts, and getting him to stop long enough for me to ask questions wasn't easy. But Bresniker presented a more practical, short-term view of The Machine.

For example, Bresniker mentioned that early versions of The Machine wouldn't necessarily be using memristors. HP Moonshot, a very large rack-mountable drawer that can contain up to 45 cartridges, is the basis for the datacenter version of The Machine. Moonshot already exists, and the cartridges currently contain Intel Atom or ARM 64-bit processors, DRAM, and storage. The storage can be hard drives or SSDs, and the drawer contains several different means for communication: a toroidal mesh, a neighbor-to-neighbor interconnect, and TCP/IP switching. Moonshot allows people to work with something that provides some of the features of The Machine today, although without the performance of memristors and photonic communications. If you read the review of the Atom-based Moonshot [5], you will see that configuring it is like working with a rack of servers in a box, where the top-of-rack switch is also in the box.

Bresniker told me that the plans are for The Machine to really eclipse Moonshot, allowing the building of thousands of heterogeneous cores connected by a photonic fabric and backed by hundreds of petabytes of NVM. The vision is an ambitious one.

HP has planned on releasing a version of Linux, called Linux++, designed to allow people to start working with an operating system that will support The Machine. Linux++ will be open source and, hopefully, a new platform for research. And there will need to be research, as having Linux working on something like The Machine, with a variety of hardware platforms as well as vast amounts of NVM arranged as a unified pool, will be very different from what researchers and developers are working with today.

Bresniker repeated a line from Fink's talk: that data will live in the peripheries, not centralized, and that we need algorithms that can access and analyze that data. When Bresniker slowed down, I asked him whether there were plans for operating systems other than Linux. Bresniker agreed that supercomputers and the Internet of Things won't be running Linux but, more likely, will be running custom operating systems, such as library OS for supercomputers and microkernels for IoT. When I asked about how to deal with crashes in a future where all memory is non-volatile, so rebooting no longer erases the problem that caused the crash, Bresniker responded that we need to build systems that are trustworthy: programs that are formally proven not to have side effects. Bresniker said that HP already has done a lot of work with fault-tolerance, and Bresniker mentioned having fault containers.

I took a couple of days to digest what I had heard from Bresniker and Fink. The Machine is an admirable concept, something that may be built, something worth doing. If HP can provide more than concepts—for example, Linux++, or better yet, working memristors—then developers and researchers will have something concrete to begin working with.

The Lineup

The opening lineup for this issue was inspired by my time at FAST '15. I encountered people presenting a poster comparing the performance of NFSv3 with NFSv4.1 in the first poster session, and asked the presenters to share their results with *;login:* readers. Chen et al. tell us about their experiments comparing both versions working on a Linux server. I think you will be pleasantly surprised, especially if you have planned on trying NFSv4.1.

I also met Christoph Hellwig during FAST, as well as during the Linux FAST workshop (see the conference report in this issue) that occurred after FAST. This wasn't the first time I had met Christoph, and I decided that I would try and uncover the path he had taken to become one of the top Linux kernel developers.

Dean Hildebrand and Frank Schmuck of IBM Research, Almaden, wrote about putting the "general" into the General Purpose File System. Their 2002 FAST paper won a Test of Time award several years ago, and they wanted to share how they had helped take a distributed file system that was designed for HPC and turn GPFS into a system that handled many different types of loads over multiple interfaces well.

I also met Carl Waldspurger during a poster session. I had already heard the presentation of their paper, where he discussed the technique they are using to calculate miss ratio curves (MRCs) that require much less time and very little memory. Not only is their work interesting, but I think that their sampling technique, using a spatially distributed hash to select their samples, could be relevant in other areas as well.

We begin our section on system administration with an article by Zbigniew Jędrzejewski-Szmek and Jóhann B. Guðmundsson on using systemd to run daemons. Systemd has become the accepted replacement in Linux for initd, the first process to start after the kernel has booted, and in this article the authors focus on the types of tasks that had traditionally been handled by a separate daemon: inetd. I learned a lot more about systemd in general, including that its capabilities are vast.

Chris Jones, Todd Underwood, and Shylaja Nukala have written about the process used for hiring SREs at Google. While this might seem an awfully narrow topic, I found myself thinking that the hiring of technical staff might function much better if managed in a fashion similar to how Google hires their SREs.

Dave Hixon and Betsy Beyer write about being a Systems Engineer (SE). If you thought becoming an SRE was tough, how about a job where there is no training or career path? I found myself wishing I were 20 years younger (for more reasons than just this), as I found the position of SE both challenging and appealing to the problem solver in me.

Andy Seely has some surreal tales from his work as an IT manager. The ideal tech manager has to balance the quirks of his reports against the oftentimes resistant framework of the larger organization. But sometimes those quirks create problems that take real creativity to solve, while at other times just a phone call helps to sort things out.

Peter Salus continues his special column on history by delving into the deep past of computer user groups. Salus starts with mainframe user groups, then takes us to the very beginning of USENIX via meetings of early UNIX users.

David Blank-Edelman and Dave Beazley synched up this issue, both writing about parallelism. Blank-Edelman shows off examples of simple parallelism in Perl, through the use of forks and threads. Dave Beazley takes a deep dive into threading in Python, with demonstrations of how the Global Interpreter Lock (GIL) affects how many threads can run and complete.

Dave Josephsen explains how to start using jmxtrans to monitor Java Virtual Machines (JVM). Dave tells us how to install, configure, and use jmxtrans so you can collect the metrics of your choice from JVMs.

Dan Geer and Dan Conway have decided to analyze the use of keywords within 12 years of articles for *IEEE Security and Privacy*. The authors search for trends, revealed through the prevalence of these keywords, that might provide hints about the security topics keeping people awake at night, as well as threats that no longer appear to be as interesting.

Robert Ferrell reflects on USENIX history, specifically, his own adventures as the SAGE historian. Warning: events that appear in his mirror may be further away than they appear.

Mark Lamourine has contributed three book reviews for this issue, on graph databases, creating visualizations within your browser, and a new book on Scratch.

We also have some conference reports in this issue. Erez Zadok, one of the FAST '15 co-chairs, wanted three of his students to have the experience of writing reports, and those reports appear in this issue. I attended Linux FAST, and did my best to capture the dialog that went on during the five-hour meeting. My goal is really to help people understand, by recounting the interaction of CS researchers and Linux kernel developers, how these two groups might work together better.

It might seem to you that I have forgotten the "electrons compute" from earlier in this column. I really haven't. I've just been saving that bit for now.

If you examine the history of computers, you might notice that early computers were not flexible at all: they were designed for performing very specific tasks. Over time, computers became more programmable and more able to work on general problems. And perhaps we have lost something by having made computers so general, as being general means that we have given up specific abilities, like the vector processors of '80s supercomputers. And perhaps we really need some of these special processors, as seen in the increasing popularity of using GPUs to complement the general purpose CPU. GPUs are, after all, special-purpose processors, but ones that never stand alone, reminding me of the floating-point coprocessors once common in early PCs.

The design described in the Tessellation paper [1] was a method of partitioning a manycore system, with heterogeneous cores, into groups based on the requirements of code executing on that group. In a way, this is not unlike one of the concepts in The Machine: heterogeneous cores connected together to form processing groups on demand.

But, by far, what I find most interesting of all about The Machine is the embrace of large amounts of non-volatile memory. Dealing with such a large pool of fast memory is quite alien to the way we in CS do things today. We've become quite accustomed to having manageable containers of memory, usually hierarchically organized, as part of servers, SAN or NAS devices, or JBODs connected to servers. In the world as imagined for The Machine, all memory forms a giant, flat space.

My guess is that we will have a difficult time wrapping our heads around such a design. Even HP spokesmen talk about moving "computation close to the data" and that "data will be distributed," which hints that there won't be a single, flat storage, but, at the very least, distributed groupings of data.

At the very least, we will be in for an interesting time indeed.

Resources

[1] Rose Liu, Kevin Klues, Sarah Bird, Steven Hofmeyr, Krste Asanović, and John Kubiatowicz; "Tessellation: Space-Time Partitioning in a Manycore Client OS": https://www.usenix.org/legacy/events/hotpar09/tech/full_papers/liu/liu_html/.

[2] Krste Asanović, "FireBox: A Hardware Building Block for 2020 Warehouse-Scale Computers," USENIX FAST '14: https://www.usenix.org/conference/fast14/technical-sessions/presentation/keynote.

[3] The Machine: http://www.hpl.hp.com/research/systems-research/themachine/.

[4] J. McCalpin, "A Survey of Memory Bandwidth and Machine Balance in Current High Performance Computers," *(circa 1995):* http://www.cs.virginia.edu/~mccalpin/papers/balance/.

[5] Tom Henderson, "First Look: HP Takes Giant Leap in Server Design" (review of Intel Atom-based Moonshot): http://www.networkworld.com/article/2174138/servers/first-look--hp-takes-giant-leap-in-server-design.html.

Is NFSv4.1 Ready for Prime Time?

MING CHEN, DEAN HILDEBRAND, GEOFF KUENNING, SOUJANYA SHANKARANARAYANA, BHARAT SINGH, AND EREZ ZADOK

Ming Chen is a PhD candidate in computer science at Stony Brook University. His research interests include file systems, storage systems, and cloud computing. He is working together with Professor Erez Zadok in building storage systems using NFS and cloud services. mchen@cs.stonybrook.edu

Dean Hildebrand manages the Cloud Storage Software team at the IBM Almaden Research Center and is a recognized expert in the field of distributed and parallel file systems. He has authored numerous scientific publications, created over 28 patents, and chaired and sat on the program committee of numerous conferences. Hildebrand pioneered pNFS, demonstrating the feasibility of providing standard and scalable access to any file system. He received a BSc in computer science from the University of British Columbia in 1998 and a PhD in computer science from the University of Michigan in 2007. dhildeb@us.ibm.com

Geoff Kuenning spent 15 years working as a programmer before changing directions and joining academia. Today he teaches computer science at Harvey Mudd College in Claremont, CA, where he has developed a reputation for insisting that students write readable software, not just working code. When not teaching or improving his own programs, he can often be found trying—and usually failing—to conquer nearby Mount Baldy on a bicycle. geoff@cs.hmc.edu.

NFSv4.1, the latest version of the NFS protocol, has improvements in correctness, security, maintainability, scalability, and cross-OS interoperability. To help system administrators decide whether or not to adopt NFSv4.1 in production systems, we conducted a comprehensive performance comparison between NFSv3 and NFSv4.1 on Linux. We found NFSv4.1 to be stable and its performance to be comparable to NFSv3. Our new (and sometimes surprising) observations and analysis cast light on the dark alleys of NFS performance on Linux.

The Network File System (NFS) is an IETF-standardized protocol to provide transparent file-based access to data on remote hosts. NFS is a highly popular network-storage solution, and it is widely supported in OSes, including FreeBSD, Linux, Solaris, and Windows. NFS deployments continue to increase thanks to faster networks and the proliferation of virtualization. NFS is also playing a major role in today's cloud era by hosting VM disk images in public clouds and providing file services in cloud storage gateways.

The continuous evolution of NFS has contributed to its success. The first published version, NFSv2, operates on UDP and limits file sizes to 2 GB. NFSv3 added TCP support, 64-bit file sizes and offsets, and performance enhancements such as asynchronous writes and `READDIRPLUS`.

The latest version, NFSv4.1, includes additional items such as (1) easier deployment with one single well-known port (2049) that handles all operations, including locking, quota management, and mounting; (2) operation coalescing via `COMPOUND` procedures; (3) stronger security using RPCSEC_GSS; (4) correct handling of retransmitted non-idempotent operations with *sessions* and Exactly-Once Semantics (EOS); (5) advanced client-side caching using *delegations*; and (6) better scalability and more parallelism with pNFS [3].

We investigated NFSv4.1's performance to help people decide whether to take advantage of its improvements and new features in production. For that, we thoroughly evaluated Linux's NFSv4.1 implementation by comparing it to NFSv3, the still-popular older version [5], in a wide range of environments using representative workloads.

Our study has four major contributions:

- a demonstration that NFSv4.1 can reach up to 1177 MB/s throughput in 10 GbE networks with 0.2–40 ms latency while ensuring fairness to multiple clients;
- a comprehensive performance comparison of NFSv3 and NFSv4.1 in low- and high-latency networks, using a wide variety of micro- and macro-workloads;
- a deep analysis of the performance effect of NFSv4.1's unique features (statefulness, sessions, delegations, etc.); and
- fixes to Linux's NFSv4.1 implementation that improve its performance by up to 11χ.

AND STORAGE

Bharat Singh is a graduate student in computer science at Stony Brook University. Before that he worked at NetApp developing storage solutions. His research interests include file and storage systems. He is working with Professor Erez Zadok in building high performance storage systems using NFS.
bharat.singh.1@stonybrook.edu

Soujanya Shankaranarayana earned her master's degree from Stony Brook University in 2014 and was advised by Professor Erez Zadok. She is presently working at Tintri Inc., a Bay Area storage startup. Her research interests involve NFS, distributed storage, and storage solutions for virtualized environments. In addition to academic research experience, she also has industry experience in distributed systems and storage systems.
soshankarana@cs.stonybrook.edu

Erez Zadok received a PhD in computer science from Columbia University in 2001. He directs the File Systems and Storage Lab (FSL) at the Computer Science Department at Stony Brook University, where he joined as faculty in 2001. His current research interests include file systems and storage, operating systems, energy efficiency, performance and benchmarking, security, and networking. He received the SUNY Chancellor's Award for Excellence in Teaching, the US National Science Foundation (NSF) CAREER Award, two NetApp Faculty awards, and two IBM Faculty awards. ezk@fsl.cs.sunysb.edu

Methodology

Hardware Configuration	
Server & Clients	Dell PowerEdge R710 (1 server, 5 clients)
CPU	Six-core Intel Xeon X5650, 2.66 GHz
Memory	64 GB
NIC	Intel 82599 10 GbE
Server Disk	RAID-0
RAID Controller	Dell PERC 6/i
Disks	Eight Intel DC S2700 (200 GB) SSDs
Read Throughput	860 MB/s
Network Switch	Dell PowerConnect 8024F
Jumbo Frames	Enabled
MTU	9000 bytes
TCP Segmentation Offload	Enabled
Round-Trip Time	0.2 ms (`ping`)
TCP Throughput	9.88 Gb/s (`iperf`)
Software Settings	
Linux Distribution	CentOS 7.0.1406
Kernel Version	3.14.17
Server File System	`ext4`
Network Settings	
NFS Implementation	Linux in-kernel
NFS Server Export Options	Default (sync set)
NFSD Threads	32
`tcp_slot_table_entries`	128
NFS Client Settings	
`rsize` & `wsize`	1MB
`actimeo`	60
NFS Security Settings	Default (no `RPCSEC_GSS`)

Table 1: Details of the experimental setup

Our experimental testbed consists of six identical machines (one server and five clients). Table 1 gives details of the hardware and software configuration.

We developed *Benchmaster*, a benchmarking framework that launches workloads on multiple clients concurrently. Benchmaster also collects system statistics using tools such as `iostat` and `vmstat`, and by reading `procfs` entries. `/proc/self/mountstats` provides particularly useful per-procedure details, including request and byte counts, accumulated queueing time, and accumulated round-trip time.

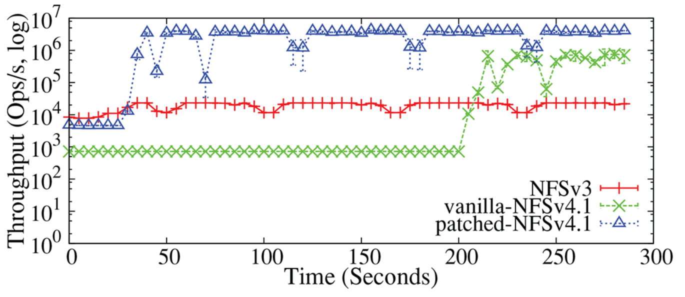


Figure 1: Aggregate throughput of five clients reading 10,000 4 KB files with 16 threads in a 10 ms-delay network ($\log_{10}$)

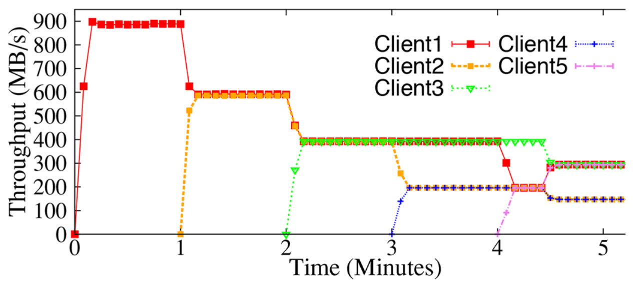


Figure 2: Sequential-read throughputs of individual clients when they were launched one after the other at an interval of one minute (results of one run of experiments)

We ran our tests long enough to ensure stable results, usually five minutes. We repeated each test at least three times, and computed the 95% confidence interval for the mean. Unless otherwise noted, we plot the mean of all runs' results, with the half-widths of the confidence intervals shown as error bars. To evaluate NFS performance over short- and long-distance networks, we injected delays ranging from 1 ms to 40 ms (Internet latency within New York State) using `netem` at the client side. For brevity, we call the network without extra delay "zero-delay," and the network with nms injected delay as "*n* ms-delay."

Major Results

We organize our major results around several questions that might arise when considering whether to adopt NFSv4.1. We hope our answers will help system administrators make well-informed deployment decisions.

Is NFSv4.1 Ready for Production?

Throughout our study, Linux's NFSv4.1 implementation functioned well and finished all workloads smoothly. This was true for three different kernel versions: 2.6.32, 3.12.0, and 3.14.17 (only the 3.14.17 results are shown here). We believe NFSv4.1's implementation to be stable, considering the wide scope and high intensity of our tests, which included dozens of micro- and macro-workloads, involved as many as 2560 threads, and reached throughput up to 1177 MB/s.

For data-intensive workloads that operate on one big NFS file, NFSv4.1 performed almost the same as the simpler NFSv3 while providing extra features. Both NFSv3 and NFSv4.1 were able to easily saturate the 10 GbE network, although we needed to enlarge the maximum TCP buffer sizes (i.e., `rmem_max` and `wmem_max`) when the network latency was long.

For metadata-intensive workloads that operate on many small files and directories, our early results showed vast differences between NFSv3 and NFSv4.1. Figure 1 shows one such result, where NFSv4.1 (the bottom curve at time 0) performed 24χ worse (note the $\log_{10}$ scale) than NFSv3 (the bottom curve at time 300) during the first 200 seconds, but jumped to be 25χ better after that. By looking at `mountstats` data and tracing the kernel with SystemTap, we found that NFSv4.1's poor early performance was due to a system bug. We fixed that with a patch [6], resulting in the upper curve (marked with triangles) in Figure 1. The performance jump at about 40 seconds was because of a new feature called *delegations*, on which we will elaborate later. For the rest of the article, we will report results of the patched NFSv4.1 instead of the vanilla version.

After the bug fix, NFSv4.1 performed close (slightly better or worse) to NFSv3 for most metadata-intensive workloads, but with exceptions in extreme settings. For example, with 512 threads per client, NFSv4.1 created small files 2.9χ *faster* or 3χ *slower* depending on the network latency.

Generally speaking, we think NFSv4.1 is almost ready for production deployment. It functioned well in our comprehensive and intense tests, although with a few performance issues in the metadata-intensive tests. However, the performance bug we found was easy to uncover, and its fix was also straightforward, suggesting that NFSv4.1 is not yet widely deployed; otherwise, these issues would already have been corrected. We argue that it is time for NFS users to at least start testing NFSv4.1 for production workloads.

Are You Affected by Hash-Cast?

Hash-Cast is a networking problem we discovered during our study; it affects not only NFS but any systems hosting concurrent data-intensive TCP flows. In our test of a sequential read workload, we frequently observed a *winner-loser pattern* among the clients, for both NFSv3 and NFSv4.1, exhibiting the following three traits: (1) the clients formed two clusters, one with high throughput (winners) and one with low throughput (losers); (2) often, the winners' throughput was approximately double that of the losers; and (3) no client was consistently a winner or a loser; a winner in one experiment might became a loser in another.

Initially, we suspected that the winner-loser pattern was caused by the order in which the clients launched the workload. To test that hypothesis, we repeated the experiment but launched the

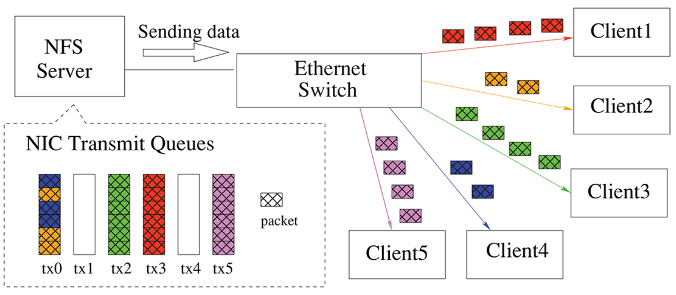


Figure 3: Illustration of Hash-Cast

clients in a controlled order, one additional client every minute. However, the results disproved any correlation between experiment launch order and winners. Figure 2 shows that Client2 started second but ended up as a loser, whereas Client5 started last but became a winner. Figure 2 also shows that the winners had about twice the throughput of the losers. We repeated this experiment multiple times and found no correlation between a client's start order and its chance of being a winner or loser.

By tracing the server's networking stack, we discovered that the winner-loser pattern is caused by the server's use of its multi-queue network interface card (NIC). Every NIC has a physical transmit queue (`tx-queue`) holding outbound packets, and a physical receive queue (`rx-queue`) tracking empty buffers for inbound packets [7]. Many modern NICs have multiple sets of `tx-queues` and `rx-queues` to allow networking to scale with the number of CPU cores, and to facilitate better NIC virtualization [7]. Linux uses hashing to decide which `tx-queue` to use for each outbound packet. However, not all packets are hashed; instead, each TCP socket has a field recording the `tx-queue` the last packet was forwarded to. If a socket has any outstanding packets in the recorded `tx-queue`, its next packet is also placed in that queue. This approach allows TCP to avoid generating out-of-order packets by placing packet *n* on a long queue and *n*+1 on a shorter one. However, a side effect is that for highly active TCP flows that always have outbound packets in the queue, the hashing is effectively done per-flow rather than per-packet.

The winner-loser pattern is caused by uneven hashing of TCP flows to `tx-queues`. In our experiments, the server had five flows (one per client) and a NIC with six `tx-queues`. If two flows were hashed into one `tx-queue` and the rest into three others, then the two flows sharing a `tx-queue` got half the throughput of the other three because all `tx-queues` were transmitting at the same rate. We call this phenomenon (i.e., hashing unevenness causing a winner-loser pattern of throughput) *Hash-Cast* (see Figure 3).

Hash-Cast explains the performance anomalies illustrated in Figure 2. First, Client1, Client2, and Client3 were hashed into `tx3`, `tx0`, and `tx2`, respectively. Then, Client4 was hashed into `tx0`, which Client2 was already using. Later, Client5 was hashed into `tx3`, which Client1 was already using. However, at 270 seconds, Client5's `tx-queue` drained and it was rehashed into `tx5`. At the experiment's end, Client1, Client3, and Client5 were using `tx3`, `tx2`, and `tx5`, respectively, while Client2 and Client4 shared `tx0`. Hash-Cast also explains why the losers usually got half the throughputs of the winners: the {0,0,1,1,1,2} distribution has the highest probability, around 69%.

To eliminate hashing unfairness, we used only a single `tx-queue` and then configured `tc qdisc` to use Stochastic Fair Queueing (SFQ), which achieves fairness by hashing flows to many software queues and sends packets from those queues in a round-robin manner. Most importantly, SFQ used 127 software queues so that hash collisions were much less probable compared to using only six. To further alleviate the effect of collisions, we set SFQ's hashing perturbation rate to 10 seconds, so that the mapping from TCP flows to software queues changed every 10 seconds.

Note that using a single `tx-queue` with SFQ did not reduce the aggregate network throughput compared to using multiple `tx-queues` without SFQ. We measured only negligible performance differences between these two configurations. We found that many of Linux's queueing disciplines assume a single `tx-queue` and could not be configured to use multiple ones. Thus, it might be desirable to use just one `tx-queue` in many systems, not just NFS servers. We have also found Hash-Cast to be related to Bufferbloat [2].

Does Statefulness Make NFSv4.1 Slow?

The biggest difference that distinguishes NFSv4.1 from NFSv3 is statefulness. Both NFSv2 and NFSv3 were designed to be stateless (i.e., the server does not maintain clients' states). A stateless server is easy to implement, but it precludes stateful tasks such as mounting, file locking, quota management, etc. Consequently, NFSv2/v3 pushed these tasks onto standalone services running on separate (and sometimes ad hoc) ports, causing maintenance problems (especially when clients need to access these ports through firewalls). Being stateful, NFSv4.1 can consolidate all its services to run on single well-known port 2049, which simplifies configuration and maintenance. However, a stateful protocol introduces overhead messages to maintain state. We ran tests to characterize this overhead.

Our tests validated that the stateful NFSv4.1 is more talkative than NFSv3. Figure 4 shows the number of requests made by NFSv4.1 and NFSv3 when we ran the Filebench File-Server workload for five minutes. For both the zero and 10 ms-latency networks, NFSv4.1 achieved lower throughput than NFSv3 despite making more requests. In other words, NFSv4.1 needs more communication with the server per file operation. In Figure 4, more than one third of NFSv4.1's requests are `OPEN` and `CLOSE`, which are used to maintain states. We also observed in

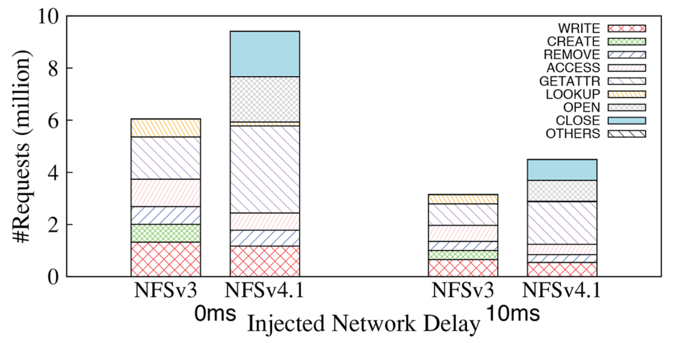


Figure 4: Number of NFS requests made by the File Server workload. Each NFSv4.1 operation represents a compound procedure. For clarity, we omit trivial and rare operations in the same compound (e.g., `PUTFH`, `SEQUENCE`, etc.).

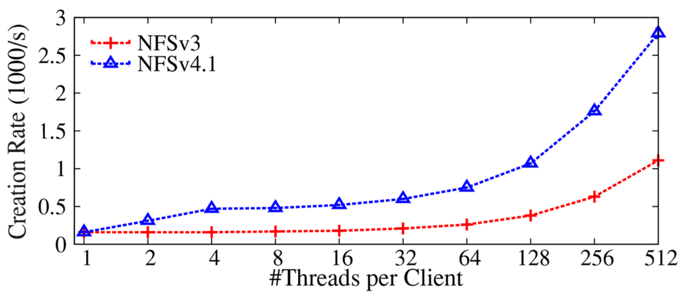


Figure 5: Rate of creating empty files in a 10 ms-delay network

other tests (particularly with a single thread and high network latency) that NFSv4.1's performance was hurt by its verbosity.

Verbosity is the result of NFSv4.1's stateful nature. To combat that effect, NFSv4.1 provides compound procedures, which can pack multiple NFS operations into one RPC. Unfortunately, compounds are not very effective: most contain only 2–4 often trivial operations (e.g., `SEQUENCE`, `PUTFH`, and `GETFH`), and applications currently have no ability to generate their own compounds. We believe that implementing effective compounds is difficult for two reasons: (1) the POSIX API dictates a synchronous programming model: issue one system call, wait, check the result, and only then issue the next call; (2) without transaction support, failure handling in multi-operation compounds is difficult.

Nevertheless, statefulness also helps performance. Figure 5 shows the speed of creating empty files in the 10 ms-delay network: NFSv4.1 increasingly outperformed NFSv3 as the number of threads grew. This is because of NFSv4.1's asynchronous RPC calls, which allow the networking layer (TCP Nagle) to coalesce multiple RPC messages. Sending fewer but larger messages is faster than sending many small ones, so NFSv4.1 achieved higher performance. Because NFSv3 is stateless, all its metadata-mutating operations have to be synchronous; otherwise a server crash might lose data. NFSv4.1, however, is stateful and can perform metadata-mutating operations asynchronously because it can restore states properly in case of server crashes.

What Is the Impact of Sessions?

Sessions are a major feature of NFSv4.1; they offer Exactly-Once Semantics (EOS). To work around network outages, NFS has evolved from UDP-only (NFSv2), to both-UDP-and-TCP (NFSv3), and now to TCP-only (NFSv4.1). However, using TCP does not solve all problems. Should a TCP connection be completely disrupted, RPC requests and replies might be lost and need to be retried once another connection is established. An NFS request might be executed more than once if (1) it was received and executed by the server, (2) its reply to the client got lost, and (3) the client retried after reconnecting. This causes problems for non-idempotent operations such as `rename`.

To solve this problem, an NFS server has a Duplicate Requests/Reply Cache (DRC), which saves the replies of executed requests. If the server finds a duplicate in the DRC, it simply resends the cached reply without re-executing the request. However, when serving an older client, the server cannot know how many replies might be lost, because clients do not acknowledge them. This means that the DRC is effectively unbounded in size.

NFSv4.1's sessions solve the problem by bounding the number of unacknowledged replies. The server assigns each session a number of slots, each of which allows one outstanding request. By reusing a session slot, a client implicitly acknowledges that the reply to the previous request using that slot has been received. Thus, the DRC size is bounded by the total number of session slots, making it feasible to keep DRC in persistent storage: for example, a small amount of NVRAM, which is necessary to achieve EOS in the face of crashes.

However, if a client runs out of slots (i.e., has reached the maximum number of concurrent requests the server allows), it has to wait until one becomes available, which happens when the client receives a reply for any of its outstanding requests. In our tests, we observed that the lack of session slots can affect NFSv4.1's performance. In Figure 6, session slots became a bottleneck when the thread count increased above 64, and thus NFSv4.1 performed worse than NFSv3. Note that the number of session slots is a dynamic value negotiated between the client and server. However, Linux has a hard limit of 160 on the number of slots per session.

How Much Does Delegation Help?

A key new feature of NFSv4.1 is delegation, a client-side caching mechanism that allows cached data to be used without lengthy revalidation. Caching is essential to good performance, but in distributed systems like NFS it creates consistency problems. NFS clients need to revalidate their cache to avoid reading stale

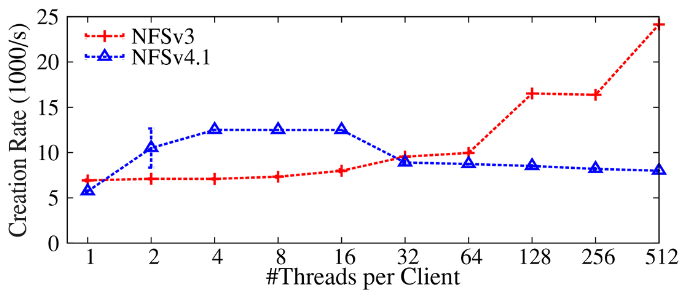


Figure 6: Rate of creating empty files in a zero-delay network

data. In practice, revalidation happens often, causing extra server load and adding delay in high-latency networks.

In NFSv4.1, the cost of cache validation is reduced by letting a server *delegate* a file to a particular client for a limited time. While holding the delegation, a client need not revalidate the file's attributes or contents. If any other clients want to perform conflicting operations, the server can recall the delegation using *callbacks*. Delegations are based on the observation that file sharing is infrequent and rarely concurrent [4]. Thus, they boost performance most of the time, but can hurt performance in the presence of concurrent and conflicting file sharing.

We studied *read delegations* (which are the only type currently supported in the Linux kernel). In Linux, a read delegation is granted if (1) the back channel to the client is working, (2) the client is opening the file with `O_RDONLY`, and (3) the file is not open for write by any client. The benefits of delegations appear in Figure 1, where they helped NFSv4.1 read small files around 172χ faster than NFSv3. In Figure 1, NFSv4.1 was simply reading from the local cache without any server communication at all, whereas NFSv3 had to send repeated `GETATTR`s for cache revalidation.

Read delegations can also improve the performance of file locking. We quantified the improvement by pre-allocating 1000 4 KB files in a shared NFS directory. Each client repeatedly opened each file, locked it, read the entire file, and then unlocked it. After ten repetitions the client moved to the next file.

Table 2 shows the number of operations performed by NFSv3 and by NFSv4.1 with and without delegation. Only NFSv4.1 shows `OPEN`s and `CLOSE`s because only NFSv4.1 is stateful. When delegations were on, NFSv4.1 used only 1000 `OPEN`s even though each client opened each file ten times. This is because each client obtained a delegation on the first `OPEN`; the following nine were performed locally. Note that NFSv4.1 did not use any `LOCK`s or `LOCKU`s because, with delegations, locks were also processed locally.

Operation	NFSv3	NFSv4.1 deleg. off	NFSv4.1 deleg. on
OPEN	0	10,001	1000
READ	10,000	10,000	1000
CLOSE	0	10,001	1000
ACCESS	10,003	9003	3
GETATTR	19,003	19,002	1
LOCK	10,000	10,000	0
LOCKU	10,000	10,000	0
LOOKUP	1002	2	2
FREE_STATEID	0	10,000	0
TOTAL	60,008	88,009	3009

Table 2: NFS operations performed by each client for NFSv3 and NFSv4.1 (with and without delegations). Each NFSv4.1 operation represents a compound procedure. For clarity, we omit trivial and rare operations in the same compound (e.g., `PUTFH`, `SEQUENCE`, `FSINFO`). NFSv3's `LOCK` and `LOCKU` (i.e., unlock) come from the Network Lock Manager (NLM).

Without a delegation (NFSv3 and NFSv4.1 with delegations off in Table 2), each application read incurred an expensive NFS `READ` operation even though the same reads were repeated ten times. Repeated reads were not served from the client-side cache because of file locking, which forces the client to invalidate the cache [1].

Another major difference among the columns in Table 2 was the number of `GETATTR`s. In the absence of delegations, `GETATTR`s were used for two purposes: to revalidate the cache upon file open, and to update file metadata upon read. The latter `GETATTR`s were needed because the locking preceding the read invalidated both the data and metadata caches for the locked file.

In total, delegations cut the number of NFSv4.1 operations by over 29χ (from 88 K to 3 K). This enabled the original stateful and "chattier" NFSv4.1 (with extra `OPEN`, `CLOSE`, and `FREE_STATEID` calls) to finish the same workload using only 5% of the requests used by NFSv3. These reductions translate to a 6–193χ speedup in networks with 0–10 ms latency.

Nevertheless, users should be aware of delegation conflicts, which incur expensive cost. We tested delegation conflicts by opening a delegated file with `O_RDWR` from another client and observed that the open was delayed for as long as 100 ms because the NFS server needed to recall outstanding delegations upon conflicts.

We have also observed that read delegations alone are sometimes not enough to significantly boost performance because writes quickly became the bottleneck, and the overall performance was thus limited. This finding calls for further investigation into how delegations can improve write performance.

Conclusions

We have presented a comprehensive performance benchmarking of NFSv4.1 by comparing it to NFSv3. Our study found that:

1. Linux's NFSv4.1 implementation is stable but suffers from some performance issues.
2. NFSv4.1 is more talkative than NFSv3 because of statefulness, which hurts NFSv4.1's performance and makes it slower in low-latency networks (e.g., LANs).
3. In high-latency networks (e.g., WANs), however, NFSv4.1 performed comparably to and even better than NFSv3, since NFSv4.1's statefulness permits higher concurrency through asynchronous RPC calls.
4. NFSv4.1 sessions can improve correctness while reducing the server's resource usage, but the number of session slots can be a scalability bottleneck for highly threaded applications.
5. NFSv4.1's read delegations can effectively avoid cache revalidation and improve performance, especially for applications using file locks, but delegation conflicts can incur a delay of at least 100 ms.
6. Multi-queue NICs suffer from the Hash-Cast problem and can cause unfairness among not only NFS clients but any data-intensive TCP flows.

Due to space limits, we have presented only the major findings and have omitted some details in explanation; please refer to our SIGMETRICS paper [1] for more results and further details.

Limitations and Future Work

Our study focused only on the performance of NFSv4.1; other aspects such as security (RPCSEC_GSS) and scalability (pNFS) should be interesting subjects to study in the future. Most of our workloads did not share files among clients. Because sharing is infrequent in the real world [4], it is critical that any sharing be representative. Finally, a different NFSv4.1 implementation than the Linux one might (and probably would) produce different results.

Acknowledgments

We thank Lakshay Akula, Vasily Tarasov, Arun Olappamanna Vasudevan, and Ksenia Zakirova for their help in this study. This work was made possible in part thanks to NSF awards CNS-1223239, CNS-1251137, and CNS-1302246.

References

[1] M. Chen, D. Hildebrand, G. Kuenning, S. Shankaranarayana, B. Singh, and E. Zadok, "Newer Is Sometimes Better: An Evaluation of NFSv4.1," in *Proceedings of the SIGMETRICS 2015*, Portland, OR, June 2015, ACM. Forthcoming.

[2] M. Chen, D. Hildebrand, G. Kuenning, S. Shankaranarayana, V. Tarasov, A. Vasudevan, E. Zadok, and K. Zakirova, "Linux NFSv4.1 Performance under a Microscope," Technical Report FSL-14-02, Stony Brook University, August 2014.

[3] D. Hildebrand and P. Honeyman, "Exporting Storage Systems in a Scalable Manner with pNFS," in *Proceedings of MSST,* Monterey, CA, 2005, IEEE.

[4] A. Leung, S. Pasupathy, G. Goodson, and E. Miller, "Measurement and Analysis of Large-Scale Network File System Workloads," in *Proceedings of the USENIX Annual Technical Conference,* Boston, MA, June 2008, pp. 213–226.

[5] A. McDonald, "The Background to NFSv4.1," *;login:*, vol. 37, no. 1, February 2012, pp. 28–35.

[6] NFS: avoid nfs_wait_on_seqid() for NFSv4.1: http://www.spinics.net/lists/linux-nfs/msg47514.html.

[7] Scott Rixner, "Network Virtualization: Breaking the Performance Barrier," *Queue,* vol. 6, no. 1, January 2008, pp. 36–37 ff.

usenix
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

The USENIX Store is Open for Business!

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

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

www.usenix.org/store

Interview with Christoph Hellwig

RIK FARROW

Christoph Hellwig has been working with and on Linux for the last ten years, dealing with kernel-related issues much of the time. In addition he is or was involved with various other open source projects. After a number of smaller network administration and programming contracts, he worked for Caldera's German development subsidiary on various kernel and user-level aspects of the OpenLinux distribution. Since 2004 he has been running his own business, focusing on consulting, training, and contracting work in the open source hemisphere. Specializing on Linux file systems and storage, he is also active in bordering areas such as virtualization and networking. He has worked for well-known customers such as Dell, SGI, IBM, Red Hat and startups like LeftHand Networks and Smapper Applied Data or Nebula. hch@infradead.org

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

I first noticed Christoph Hellwig during FAST conferences. With his spiky hair and non-academic background, he had a different presence from most attendees. But I could tell he was an expert when it came to XFS, a file system that was created by Silicon Graphics specifically for working with large files, then later ported to Linux.

Since we both attended Linux FAST, I got to know Christoph a bit better. Christoph is an independent consultant and a Linux kernel developer, a combination that is rare, as most Linux developers have some corporate sponsorship, usually in the form of a job. I also noticed that Christoph likes to ski, reminding me of the many ski bums I've met who arrange their life and work so they are free to ski when the slopes are best.

But unlike those ski bums, Christoph has established his chops as a Linux file system and kernel developer expert. I was curious about how he got to his unusual position, and started this interview after watching Christoph help run the Linux FAST '15 workshop (see summary in this issue).

Rik: When/how did you start working with Linux?

Christoph: I was in high school and learned about Linux from a magazine related to all things Internet that was popular in Germany in those days, and came with a ready-made Linux install CD. I installed it on my Dad's PC. Eventually, I got my own PC and had to fix the Linux sound driver for it, which was my first kernel contribution.

Rik: You worked for a while at Caldera, then started taking classes in Austria.

Christoph: I only worked for Caldera for about a year, maybe a year and a half, and only did so part time while finishing high school. Apparently I was productive enough that no one really knew I was a high school student. When Caldera shut down their German engineering office I got an offer from SGI to work full time for them out of their Munich office.

Math was interesting but a lot of work, so I switched to computer science. That turned out to be a bad idea, as I already had a background in computer science and now a fair knowledge of math, so the fairly low standards of the CS curriculum started to bore me. I was much happier working a real job where I learned from reading code, talking (or emailing) with experts in their field, and reading the bits of CS research that actually matter to me, which I wasn't exposed to at all at my university.

Through a loophole, I was taking lots of masters-level courses even though I never got an undergrad degree, but abusing those loopholes made me lose a lot of credits whenever they changed the curriculum, so in the end I had to give up my plan to slowly get my undergrad degree while working full time (and skiing full time!). In retrospect, trying to get that degree at all is probably the biggest regret I have, but that's definitively not the feeling I had back then.

Rik: When did you start working with file systems? Or how did you get involved? The first time I recall seeing you was at FAST, and you were making points about XFS. In my mind, I found myself labeling you the XFS guy.

Christoph: I started out my "file-system career" by hacking on support for simple foreign file systems early on in my Linux career—that's where my maintainer hat for the SysV / V7 file-system driver started as well. Then I moved on to play with the newly open-sourced IBM JFS file system and helped to clean it up so it could be merged into the Linux kernel. SGI hired me as junior file-system person with Linux community experience in 2002, and I gradually moved up the food chain from there.

Rik: During Linux FAST 2015, you and Ted Ts'o often mentioned tools that actually help make it easier to develop concurrent code inside of the kernel, rather than using FUSE. What are these tools, and how does one learn about using them?

Christoph: There are a few tools I rely on a lot:

- lockdep
  This is a tool to verify locking order, but also much more complicated constraints, such as locks taken in the page-out path but not held around memory allocation, which can lead to page outs. It also detects locks that are held while the containing objects are also holding locks, and much more. In a heavily multithreaded environment, this tool probably has the checks that code triggers most often.
- slab debugging
  The instrumented memory allocator that detects all kinds of use after frees is highly useful, although there are lots of libraries that provide the same functionality in user space these days. There also are the kmemleak and kasan tools that allow for very advanced memory leak detection.
- instrumented linked list / debug object
  We also have the instrumented version of linked lists, which is very helpful in avoiding corruption, and the debug object's functionality that allows you to enforce lifetime rules for arbitrary kernel objects.
- static trace points (trace events)
  Linux now has a very nice way of embedding trace points into code that has almost zero overhead if not enabled. Code that has a lot of those is really easy to instrument, especially since the tools for it allow nice use cases like only enabling the events for specific processes or various other conditions. The trace events also double as software profiling counters for the perf tool, allowing for very interesting semantic profiles.

But even more important than these are tools that help improve the code at runtime, most importantly the "sparse" tool, which allows us to, for example, annotate integer types as "big endian" or "little endian" and warn about missing conversions without forcing the use of complex types.

All the kernel debug options are documented sparsely in their help texts and the kernel documentation. For information more suitable to beginners, a Web search usually gives good results on sites such as stackoverflow.com.

Rik: Do you think it is harder to get involved as a Linux kernel developer now than it was when you started?

Christoph: I think doing the same work is easier these days with all the tools we have, but finding it might be harder, and becoming relevant definitively is.

Rik: You said you work a lot on the SCSI subsystem. Why is the SCSI subsystem so important to file systems?

Christoph: SCSI is one of the two standards most block storage devices speak today. SCSI is used over SAS, Fibre Channel, and iSCSI for enterprise storage; over USB for many consumer devices; and these days also in a lot of virtualized environments. Many operating systems, including Linux, also go through the SCSI layer to access the other major protocol, ATA, for historical reasons. In short, SCSI is what most block-based file systems end up using underneath, and it is one of the most important layers below the file system.

Rik: Your name was in the news in March 2015. You are involved in a legal action against VMware, because VMware has used Linux code in their hypervisor, as well as used Busybox as a shell [1, 2]. I'm wondering why you, someone who is self-employed and thus only making money when doing billable work, is taking the time to be involved with this lawsuit?

Christoph: Mostly, I want to avoid people piggybacking on my work, or the work of us kernel developers in general, without giving anything back.

Resources

[1] Software Freedom Conservancy, "Frequently Asked Questions about Christoph Hellwig's VMware Lawsuit": http://sfconservancy.org/linux-compliance/vmware-lawsuit-faq.html.

[2] Jonathan Corbet, "A GPL-Enforcement Suit against VMware: https://lwn.net/SubscriberLink/635290/e501ce0264c182f4/.

On Making GPFS Truly General

DEAN HILDEBRAND AND FRANK SCHMUCK

Dean Hildebrand manages the Cloud Storage Software team at the IBM Almaden Research Center and is a recognized expert in the field of distributed and parallel file systems. He pioneered pNFS, demonstrating the feasibility of providing standard and scalable access to any file system. He received a BSc degree in computer science from the University of British Columbia in 1998 and a PhD in computer science from the University of Michigan in 2007. dhildeb@us.ibm.com

Frank Schmuck joined IBM Research in 1988 after receiving a PhD in computer science from Cornell University. He is a Distinguished Research Staff Member at IBM's Almaden Research Center, where he serves as a Technical Leader of the Parallel File Systems group and Principal Architect of IBM's General Parallel File System (GPFS). His research interests include storage systems, distributed systems, and fault tolerance. fschmuck@us.ibm.com

GPFS (also called IBM Spectrum Scale) began as a research project that quickly found its groove supporting high performance computing (HPC) applications [1, 2]. Over the last 15 years, GPFS branched out to embrace general file-serving workloads while maintaining its original distributed design. This article gives a brief overview of the origins of numerous features that we and many others at IBM have implemented to make GPFS a truly general file system.

Early Days

Following its origins as a project focused on high-performance lossless streaming of multimedia video files, GPFS was soon enhanced to support high performance computing (HPC) applications, to become the "General Parallel File System." One of its first large deployments was on ASCI White in 2002—at the time, the fastest supercomputer in the world. This HPC-focused architecture is described in more detail in a 2002 FAST paper [3], but from the outset an important design goal was to support general workloads through a standard POSIX interface—and live up to the "General" term in the name.

An early research prototype was based on an architecture similar to today's HDFS and pNFS, with a single server storing directories and handling metadata operations, redirecting reads and writes to a separate set of data servers. While scaling well for sequential I/O to large files, performance of metadata operations and small file workloads was typically worse—or at least no better—than for a traditional file server. In other words, this early prototype did not do the G in GPFS justice.

The improved design, which largely remains to this day, eliminates the single server bottleneck by managing both data and metadata in a fully distributed fashion across the whole cluster. Both data and metadata are stored on shared storage devices that are equally accessible from all cluster nodes. A distributed lock manager coordinates access to the file system, implements a cache consistency protocol, and provides necessary synchronization for proper POSIX semantics of individual file system operations. This allows each node to safely modify metadata directly on disk instead of going through a separate metadata server. Over the years, this original design has proven flexible enough to support numerous other application domains, such as cloud computing, network attached storage (NAS), and analytics, as shown in Figure 1.

The Basics

Since its beginnings, GPFS has been deployed on a wide range of cluster types and sizes, with the larger clusters serving the scientific computing needs of national research laboratories, small-to-medium-0sized clusters serving commercial HPC applications (e.g., oil exploration and engineering design), and clusters as small as two nodes, where GPFS may be used primarily for its fault-tolerance rather than scaling abilities (e.g., highly available database server). The original design was targeted at storage area networks (SANs). Support for network shared disk (NSD) access over TCP/IP and eventually InfiniBand via dedicated I/O server nodes was added for increased scalability and flexibility. This then also enabled support for

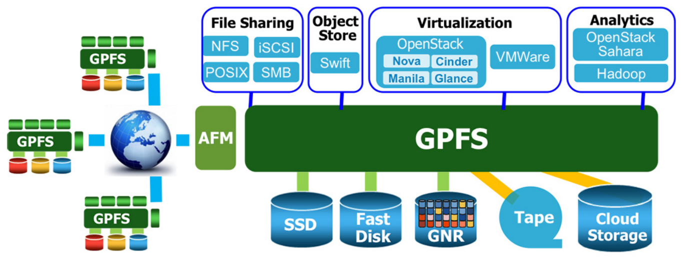


Figure 1: IBM Research has prototyped the use of GPFS with numerous APIs and storage devices within a single namespace, including the use of Active File Management (AFM) to share data across WANs.

commodity clusters consisting of server nodes with internal disks and SSDs.

The distributed locking architecture is a good match for scalable, general file-serving applications, especially for workloads consisting of a large collection of independent working sets (e.g., different users accessing different sets of files). Once an application has collected lock tokens that cover its working set from the distributed lock manager, it can read, cache, and update all data and metadata it needs independently, without any further interaction with other cluster nodes. In this manner, file access in GPFS can be just as efficient as a local file system, but with the ability to scale out by adding nodes and storage devices to meet growing bandwidth and capacity demands.

The first major feature added to GPFS for general, non-HPC workloads were read-only file system snapshots in 2003. This is particularly useful for serving a large user data set, since it allows an individual user to retrieve accidentally deleted files without requiring administrator assistance. Initially limited to 32 file system snapshots, the feature was later expanded to larger numbers and finer-grained snapshots, including writable snapshots of individual files.

At the same time, GPFS expanded its reach by extending its OS and hardware support from AIX on IBM Power servers to Linux on x86 and Power and later to Windows and Linux on mainframes. While the initial Linux release in 2001 did not allow mixing AIX and Linux in a single cluster, full heterogeneous cluster support was added a couple years later.

Larger and more diverse clusters also required continuing improvements in cluster management infrastructure. In 2004, the external cluster manager used early on was replaced with a built-in cluster manager using a more-scalable, hierarchical architecture. A subset of designated "quorum nodes" is responsible for ensuring system integrity by electing a unique cluster leader, monitoring the status of all other nodes in the cluster, and driving recovery in response to node failures. In 2007, support was added for rolling upgrades, allowing GPFS software to be upgraded one node at a time without ever shutting down the whole cluster, a critical feature for both HPC and general computing alike. Other features added in subsequent years, or actively being developed, include extended file attributes, a scalable backup solution, encryption, and compression.

Protecting Data, the Crown Jewels

The need for advanced data protection first became apparent in the HPC context, but growing data volumes means that reliability issues previously only seen in very large clusters now affect general-purpose storage systems as well. Experiences with traditional RAID controllers in the GPFS deployment for the ASC Purple supercomputer at Lawrence Livermore National Laboratory in 2005 had shown that when aggregating ~10,000 disks into a single system, very rare failure events become frequent enough that in these systems partial data loss became a real possibility. These included double disk failures, loss of a RAID stripe due to checksum errors during rebuild, off-track writes, and dropped writes. Furthermore, since disk failures were constantly occurring and their rebuild times were taking longer due to increased disk capacity, the whole system was being slowed down.

To eliminate the drawbacks of hardware storage controllers, in 2011 GPFS introduced an advanced, declustered software RAID algorithm integrated into its I/O servers, called GPFS Native RAID (GNR) [4]. Apart from simple replication, GNR offers a choice of Reed-Solomon erasure codes that tolerate up to three concurrent failures. Data, parity, and spare space are distributed across large numbers of disks, speeding up rebuild times with minimal impact on the foreground workload. Write version numbers and end-to-end checksums allow GNR to detect and recover from lost or misdirected writes, and care is taken to ensure related erasure code strips are placed in separate hardware failure domains, e.g., disk drawers, to improve availability. A background scrubbing process verifies checksum and parity values to detect and fix silent disk corruption or latent sector errors before additional errors might render them uncorrectable. The current implementation relies on a conventional, dual-ported disk enclosure filled with disks in a JBOD ("just a bunch of disks") configuration to provide redundant paths to disk in order to handle a failure of one of its primary I/O servers by a designated backup server. A current research project is exploring the use of internal disks by spreading data and parity across disks in different server nodes (network RAID).

Now that GPFS no longer relies on storage controller hardware, support was added for other "typical" storage controller features, including the ability for data to be replicated across different geographical locations for disaster recovery purposes. For shorter distances, synchronous replication is performed via standard GPFS data replication by creating a cluster that stretches across nodes at multiple sites. For larger distances, GPFS Active File Management (AFM), which was originally designed for file caching across wide area networks, can be configured to asynchronously replicate files between two file systems at separate sites [5].

Pooling Your Data Without Getting Wet

In 2003, IBM introduced its Total Storage SAN File System (SAN-FS) as a "file virtualization" solution for storage area networks. From the outset, it was aimed at general commercial applications, but soon also branched out into data-intensive applications. By 2005, it became apparent that SAN-FS and GPFS catered to increasingly overlapping market segments, and IBM started an effort to merge the two product lines. This lead to GPFS adopting some of the unique features of SAN-FS, including native Windows support and, most notably, Information Lifecycle Management (ILM) through the concepts of storage pools and filesets [6].

Storage pools are a means of partitioning the storage devices that make up a file system into groups with similar performance and reliability characteristics. User-defined "placement policy" rules allow assigning each file to a storage pool so as to match application requirements to the most appropriate and cost-effective type of storage. Periodically evaluated "management policy" rules allow migrating files between pools as application requirements change during the lifecycle of a file. Policy rules may also change file replication; delete files; invoke arbitrary, user-specified commands on a selected list of files; or migrate rarely accessed data to an "external pool" for archival storage. The policy language allows selecting files based on file attributes, such as its name, owner, file size, and timestamps, as well as extended attributes set by the user. Data migrated to external storage either via policy or a traditional external storage manager is recalled on demand using the standard Data Management API (DMAPI).

Filesets provide a way to partition the file system namespace into smaller administrative units. For example, the administrator may define user and group quotas separately for each fileset or place limits on the total amount of disk space occupied by files in each fileset. GPFS also allows creating snapshots of individual filesets instead of a whole file system. Filesets also provide a convenient way to refer to a collection of files in policy rules.

Three Amigos: NFS, SMB, and Object

A parallel file system provides a powerful basis for building higher-level scalable, fault-tolerant services by running a service instance on each cluster node. Since all nodes have equal access to all file system content, an application workload can be distributed across the cluster in very flexible ways, and if one node fails, the remaining nodes can take over. This is easiest to implement for services that do not need to maintain any state outside of the file system itself. The canonical example is an NFS file server, due to the stateless nature of the NFSv3 protocol: servers running on different nodes in the cluster can simply export the same file system without requiring any additional coordination among the different servers. For client-side data caching, the NFS protocol relies on file modification time (`mtime`) maintained by the file system, but since mtime is not critical for HPC applications, GPFS only provided an approximate mtime value with eventual consistency semantics. This was soon fixed since approximate mtime is not sufficient to guarantee NFS close-to-open consistency semantics: if a reader opens a file after a writer has closed it, the reader should see the new file content.

An example of one of the first systems exploiting GPFS capabilities to provide a scalable file server solution is the Global Storage Architecture (GSA) service deployed within IBM starting in 2002. This replaced the existing AFS and DCE-based infrastructure, and is still actively used within IBM worldwide today. To help customers implement similar solutions, we added a "clustered NFS" (CNFS) feature to the base product, which manages NFS server failover and failback, including IP address takeover and `lockd` recovery.

While NFSv3 was nominally stateless, support for richer, stateful protocols like NFSv4 and the Windows Server Message Block (SMB) make it harder to turn a single server into a scalable, clustered solution. The simplest approach is to partition the namespace across the cluster and let only one node at a time serve files under each directory subtree. This avoids complexity, but limits load balancing since a "hot" subtree may overload its assigned node. A better approach is to add a clustering layer for managing distributed, protocol-specific state above the file system. The Clustered Trivial Database (CTDB) is just such a layer, developed in collaboration with the open source community, which integrates Samba servers running on different nodes within a cluster into a single, scalable SMB server. A scalable NFS and SMB file-serving solution based on this technology was made available as an IBM service offering in 2007 and as a NAS appliance in 2011.

One downside with layering protocol-specific cluster managers on top of a parallel file system is a lack of coordination between different protocols. For example, a file lock granted to an SMB client will not be respected by a client accessing the same file over NFS or by an application running on one of the nodes in the cluster accessing the file directly. So a third approach to implementing richer services is to add functionality to the file system for maintaining protocol-specific state consistently across the cluster. By taking advantage of features originally added for the GPFS Windows client, such as a richer ACL model and extensions to the GPFS distributed lock manager to implement Windows share-modes, the NFS server can implement features such as delegations, open modes, and ACLs—without a separate clustering layer. An immediate advantage is better coordination between NFS clients and file access via SMB, FTP, and HTTP protocols.

In 2014, GPFS provided support for OpenStack Swift [7], which provides a stateless clustered layer for REST-based data access through protocols such as Swift and S3. Object storage systems have a lot in common with HPC, as they tend to have a large capacity (several PBs and larger), have a high number of simultaneous users, and span multiple sites. Many GPFS features have a direct benefit in this new domain, including scalable file creation and lookup, data tiering and information lifecycle management, GNR software-based erasure coding, and snapshots. Support for Swift does much more than provide a simplified method for data access; it includes several new features such as secure and multi-tenant access to data, role-based authentication, REST-based storage management, and a simplified flat namespace. All objects are stored as files, which enables native file access (e.g., Hadoop, NFS, SMB, POSIX) to objects without a performance-limiting gateway daemon. This capability means that objects within GPFS aren't in their own island of storage, but are integrated into a user's entire workflow.

Branching out from HPC to NAS and object storage provided an impetus for numerous improvements in GPFS to handle small-file and metadata-intensive workloads more efficiently. Allowing data of small files to be stored in the file inode instead of a separate data block improves storage efficiency and reduces the number of I/Os to access small file data. Keeping metadata for a large number of files cached in memory proved vital for good NAS performance, but put a greater load on the token server for each file system. GPFS therefore introduced a new token protocol that uses consistent hashing to distribute tokens for all file systems across any number of nodes in the cluster. Since GPFS records metadata updates in a per-node recovery log, metadata commits can be sped up by placing recovery logs in a dedicated storage pool of fast storage devices. As wider use of fast storage devices eliminates the devices itself as the performance limiting factor, the efficiency of the file system software stack as a whole becomes increasingly important, with particular attention required to minimizing overhead for synchronizing access to in-memory data structures on modern NUMA architectures.

Cloudy with a Chance of High-Performance

In 2014, there was a shift towards delivering open systems and software-defined-storage to customers. This shift was primarily motivated by customers frustrated with vendor lock-in, lack of ability to customize a solution, and also the desire (primarily for cost reasons) to leverage commodity hardware.

The OpenStack project fits well with this new way of thinking, offering an open cloud management framework that allows vendors to plug into myriad APIs. Beyond supporting the Swift object storage layer discussed previously, we have integrated support for Nova (which provisions and manages large networks of VMs), Glance (the VM image repository), and Cinder (which provides block-based storage management for VMs). Most recently, we delivered a driver for Manila, which allows users to provision file-based data stores to a set of tenants, and we are currently investigating support for Sahara, the easy to use analytics provisioning project.

Initially, there was some concern that using a file system for all of these services was not a good fit, but we found the more we integrate GPFS with all of the OpenStack services, the more benefits arise from using a single data store. Workflows can now be implemented (and automated) where files and data stores are provisioned and utilized by applications and VMs with zero-data movement as the workflow shifts from one management interface to the next. In another example, AFM can be leveraged to build a hybrid cloud solution by migrating OpenStack data to the cloud and then back again as needed.

Virtualization, and its use in the cloud, has also introduced a relatively new I/O workload that is much different than both NAS workloads, which primarily perform metadata-intensive operations, and HPC workloads, which do large writes to large files. VMs write small, random, and synchronous requests to relatively large (8–80 GB) disk image files [8]. To support this workload, we implemented a Highly Available Write Cache (HAWC), which allows buffering of small VM writes in fast storage, allowing them to be gathered into larger chunks in memory before being written to the disk subsystem. Further, we increased the granularity at which GPFS tracks changes to a file to avoid unnecessary read-modify-write sequences that never occurred previously in HPC workloads.

Moving forward, as public and private clouds continue to emerge, and more and more applications make the transition (including HPC applications), new requirements are emerging above and beyond being able to deliver high performance data access. One area that has a much different model from HPC is security and management. The "trusted root" model common in HPC datacenters is rarely acceptable, replaced by a more fine-grained and scalable role-based management model that can support multiple tenants and allow them to manage their own data. For management, supporting a GUI and REST-API is no longer just nice to have, as is an audit log for retracing operations performed on the system. As well, scaling existing monitoring tools and delivering higher-level insights on system operation will be key features of any cloud storage system.

Another area of interest is data capacity, where HPC has traditionally led the way, but cloud computing is catching up and is possibly poised to overtake HPC in the near future. For example, some cloud datacenters are scaling by up to 100 PB per year. The challenge for GPFS is less about figuring out how to store all that data (the GPFS theoretical limit on the number of files in a single file system is 2^{64}, after all), but more about providing a highly available system at scale that can be efficiently managed. For the issue of storing trillions of files, the classical hierarchical file-system directory structure, no matter how scalable, is not an intuitive method for organizing and finding data. GPFS support for object storage improves this by introducing both a simpler flat namespace as well as a database for faster indexing and searching. For the issue of high availability, the impact of catastrophic failure must be limited at any level of the software and hardware stack. To do this, failure domains must be created at every level of the software stack, including the file system, such that when a catastrophic failure occurs in one failure domain, the remaining failure domains remain available to users.

Analyze This (and That)

When analytics frameworks like Hadoop (and its file system HDFS) started, they focused on a specific class of problems that exploited locality to scale I/O bandwidth. So to support analytics, a Hadoop connector was implemented and a few key changes were made to GPFS to support storage rich servers. First, we increased the maximum replication from two to three, which was primarily a testing effort, and ensured one of those replicas was stored on the local server. Second, the traditional parallel file system method of striping small (1 MB) chunks across the entire cluster would overflow network switches, so block groups were introduced to allow striping in much larger chunks (128 MB). Third, failure groups were extended to understand network hierarchies, instead of just the flat networks common in HPC.

Recently, a shift has occurred that brings new life to running analytics on the original GPFS architecture. The combination of cheaper, fast networks with the emergence of new analytic workloads such as Hive and HBase mitigates the benefit of data locality in many cases. These workloads perform smaller and more random I/O, benefiting from the scalable metadata and optimized data path in GPFS. In addition, support for POSIX semantics (and therefore in-place updates) allows a wide range of such analytics workloads to be developed.

Conclusion

GPFS represents a very fruitful and successful collaboration between IBM Research and Product divisions, with customer experiences providing a rich source of interesting and challenging research problems, and research helping to rapidly bring advanced technology to the customer. Living up to the G in GPFS has thus been a fun if not always an easy or straightforward journey.

Looking ahead, GPFS will continue to evolve and strengthen its support for all types of enterprise workloads, enabling users to have a single common data plane (aka "data lake") for all of their application requirements. In HPC, GPFS has recently been chosen as the file system in two petaflop supercomputers set to go online in 2017 [9], whose "data-centric" design is a milestone in the path towards exascale computing. Simultaneously, GPFS's trip into the cloud is yielding exciting new features and functionality addressing new and evolving storage needs.

References

[1] IBM Spectrum Scale: www-03.ibm.com/systems/storage/spectrum/scale.

[2] D. Hildebrand, F. Schmuck, "GPFS," in Prabhat and Q. Koziol (eds), *High Performance Parallel I/O* (Chapman and Hall/CRC, 2014), pp. 107–118.

[3] F. Schmuck, R. Haskin, "GPFS: A Shared-Disk File System for Large Computing Clusters," in *Proceedings of the 1st USENIX Conference on File and Storage Technologies (FAST '02),* 2002: https://www.usenix.org/legacy/events/fast02/schmuck.html.

[4] IBM, GPFS Native RAID Version 4 Release 1.0.5, Administration, 2014: http://publib.boulder.ibm.com/epubs/pdf/c2766580.pdf.

[5] M. Eshel, R. Haskin, D. Hildebrand, M. Naik, F. Schmuck, R. Tewari, "Panache: A Parallel File System Cache for Global File Access," in *Proceedings of the Eighth USENIX Conference on File and Storage Technologies (FAST '10),* 2010: https://www.usenix.org/legacy/events/fast10/tech/full_papers/eshel.pdf.

[6] IBM, General Parallel File System Version 4 Release 1.0.4, Advanced Administration Guide, (SC23-7032-01), 2014: http://publib.boulder.ibm.com/epubs/pdf/c2370321.pdf.

[7] IBM, "A Deployment Guide for IBM Spectrum Scale Object": http://www.redbooks.ibm.com/abstracts/redp5113.html?Open.

[8] Vasily Tarasov, Dean Hildebrand, Geoff Kuenning, Erez Zadok, "Virtual Machine Workloads: The Case for New Benchmarks for NAS," in *Proceedings of the Eleventh USENIX Conference on File and Storage Technologies (FAST '13),* 2013: https://www.usenix.org/conference/fast13/technical-sessions/presentation/tarasov.

[9] IBM and Nvidia to Build 100 Petaflop Supercomputers, November 2014: http://www.vrworld.com/2014/11/14/ibm-and-nvidia-to-build-100-petaflop-supercomputers.

usenix
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

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 conference proceedings and videos. We stand by our mission to foster excellence and innovation while supporting research with a practical bias. Your financial support plays a major role in making this endeavor successful.

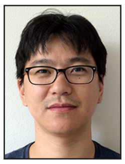
Please help to us to sustain and grow our open access program. Donate to the USENIX Annual Fund, renew your membership, and ask your colleagues to join or renew today.

www.usenix.org/annual-fund

Beyond Working Sets
Online MRC Construction with SHARDS

CARL A. WALDSPURGER, NOHHYUN PARK, ALEXANDER GARTHWAITE, AND IRFAN AHMAD

Carl Waldspurger has been conducting research at CloudPhysics since its inception. Carl has a PhD in computer science from MIT. His research interests include resource management, virtualization, security, data analytics, and computer architecture.
carl@cloudphysics.com



Nohhyun Park is a Software Engineer at CloudPhysics working on data analytics and the supporting pipeline. He has a PhD in electrical and computer engineering from the University of Minnesota and is interested in workload characterization and performance modeling for large-scale systems.
nohhyun@cloudphysics.com

Alexander Garthwaite is a Software Engineer at Twitter and an advisor to CloudPhysics. Alex has a PhD in computer and information science from the University of Pennsylvania. His interests include resource management, virtualization, programming language implementation, and computer architecture.
alex@cloudphysics.com

Irfan Ahmad is the CTO and cofounder of CloudPhysics. Irfan works on interdisciplinary endeavors in memory, storage, CPU, and distributed resource management. He has published at ACM SOCC (best paper), USENIX ATC, FAST, IISWC, etc. He has chaired HotCloud, HotStorage, and VMware's Innovation Conference.
irfan@cloudphysics.com

Estimating the performance impact of caching on storage workloads is an important and challenging problem. Miss ratio curves (MRCs) provide valuable information about cache utility, enabling efficient cache sizing and dynamic allocation decisions. Unfortunately, computing exact MRCs is too expensive for practical online applications. We introduce a novel approximation algorithm called SHARDS that leverages uniform randomized spatial sampling to construct surprisingly accurate MRCs using only modest computational resources. Operating in constant space and linear time, SHARDS makes online MRC generation practical for even the most constrained computing environments.

Caches designed to accelerate data access by exploiting locality are pervasive in modern systems. Operating systems and databases maintain in-memory buffer caches containing "hot" blocks considered likely to be reused. Server-side or networked storage caches using flash memory are popular as a cost-effective way to reduce application latency and offload work from rotating disks. Virtually all storage devices—ranging from individual disk drives to large storage arrays—include significant caches composed of RAM or flash memory.

Since cache space consists of relatively fast, expensive storage, it is inherently a scarce resource and is commonly shared among multiple clients. As a result, optimizing cache allocations is important. Today, administrators or automated systems seeking to optimize cache allocations are forced to resort to simple heuristics, or to engage in trial-and-error tests. Both approaches to performance estimation are problematic.

Heuristics simply don't work well for cache sizing, since they cannot capture the temporal locality profile of a workload. Without knowledge of marginal benefits, for example, doubling (or halving) the cache size for a given workload may change its performance only slightly, or by a dramatic amount.

Trial-and-error tests that vary the size of a cache and measure the effect are not only time-consuming and expensive, but also present significant risk to production systems. Correct sizing requires experimentation across a range of cache allocations; some might induce thrashing and cause a precipitous loss of performance. Long-running experiments required to warm up caches or to observe business cycles may exacerbate the negative effects. In practice, administrators rarely have time for this. Resigned to severe imbalances in cache utility, they often end up buying additional hardware.

The ideal approach is estimating workload performance as a function of cache size by modeling its inherent temporal locality; in other words, by incorporating information about the reuse of blocks. As the workload accesses each individual block, its *reuse distance*—the number of other unique intervening blocks referenced since its previous use—is captured and accumulated in a histogram. The complete *miss ratio curve* (MRC) for a workload is

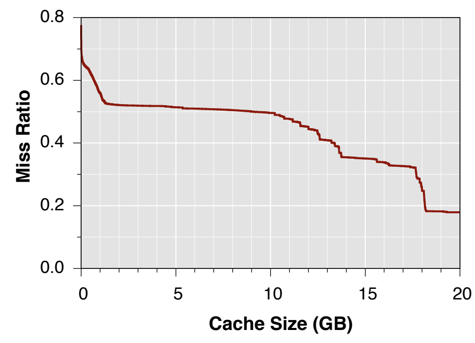


Figure 1: Example MRC. A miss ratio curve plots the ratio of cache misses to total references, as a function of cache size. Lower is better.

computed directly from its reuse-distance histogram. Unfortunately, even the most efficient exact implementations for MRC construction are too heavyweight for practical online use in production systems.

Figure 1 shows an example MRC, which plots the ratio of cache misses to total references for a workload (y-axis) as a function of cache size (x-axis). The higher the miss ratio, the worse the performance; the miss ratio decreases as cache size increases. MRCs come in many shapes and sizes, and represent the historical cache behavior of a particular workload. This particular MRC reveals a staircase pattern representing knees in the working set: the first 2 GB of cache provide a large improvement, followed by a flat region for the next 8 GB, then another dropoff, and so on. Cache performance is highly nonlinear, so identifying such knees is critical for making efficient allocation and partitioning decisions.

Assuming some level of stationarity in the workload pattern at the time scale of interest, the workload's MRC can be used to predict its future cache performance. An administrator can use a system-wide miss ratio curve to help determine the aggregate amount of cache space to provision for a desired improvement in overall system performance. Similarly, an automated cache manager can utilize separate MRCs for multiple workloads of varying importance, optimizing cache allocations dynamically to achieve service-level objectives.

MRC Construction

In their seminal paper, Mattson, Gecsei, Slutz, and Traiger [1] proposed a technique to generate models of behavior for all cache sizes in a *single pass*. Since then, Mattson's technique has been applied widely. However, the computation and space required to generate such MRCs have been prohibitive. For a trace of length N containing M unique references, the most efficient exact implementations of this algorithm have an asymptotic cost of $O(N \log M)$ time and $O(M)$ space [4].

Given the nonlinear computation cost and unbounded memory requirements, it is impractical to perform real-time analysis in production systems. Even when processing can be delayed and performed offline from a trace file, memory requirements may still be excessive. For example, we have collected many traces for which conventional MRC construction does not fit in 64 GB RAM. This is especially important when modeling large storage caches; in contrast to RAM-based caches, affordable flash cache capacities often exceed 1 TB, requiring many gigabytes of RAM for traditional MRC construction.

The limitations of existing MRC algorithms led us to consider a very simple idea. What if we place a filter in front of a conventional MRC algorithm to randomly sample only a small subset of its input blocks, and run the full algorithm over these samples? The question was whether or not this would be sufficiently efficient and accurate for practical use.

Our answer to this question is a new algorithm based on spatially hashed sampling called *SHARDS* (*S*patially *H*ashed *A*pproximate *R*euse *D*istance *S*ampling) [7]. SHARDS runs in constant space and linear time by tracking only references to representative locations, selected dynamically based on a function of their hash values.

Randomized spatial sampling allows SHARDS to use several orders of magnitude less space and time than exact methods, making it inexpensive enough for practical online MRC construction in high-performance systems. The dramatic space reductions also enable analysis of long traces that is not feasible with exact methods. Traces that consume many gigabytes of RAM to construct exact MRCs require less than 1 MB for accurate approximations. The low cost even enables concurrent evaluation of different cache configurations (e.g., block size or write policy) using multiple SHARDS instances.

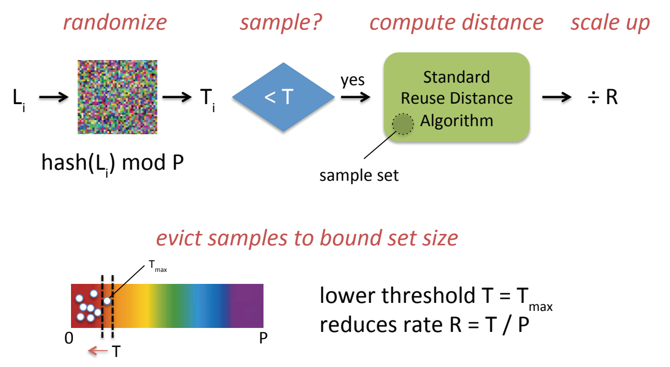


Figure 2: SHARDS algorithm overview. SHARDS filters the input to a standard reuse-distance algorithm using spatially hashed sampling. Each input location L_i is mapped to a hash value T_i, which is compared to a global threshold T that determines the sampling rate R. The threshold is lowered progressively as needed to maintain a fixed bound on the size of the sample set, s_{max}.

SHARDS Algorithm

The SHARDS algorithm, shown in Figure 2, is conceptually simple. A hash function takes each referenced location L_i, such as a logical block number (LBN), and maps it to a hash value T_i, that is uniformly distributed over the range [0, P), depicted as painting each location with a random color.

A global threshold T is used to divide the hash value space into two partitions, or "shards." Locations that hash to values below the threshold are sampled, and others are filtered out. The sampling rate R is simply the fraction of the hash value space that is sampled. In practice, typical sampling rates are significantly lower than 1%. More generally, using the sampling condition *hash(L)* mod $P < T$, with modulus P and threshold T, the effective sampling rate is $R = T/P$, and each sample represents $1/R$ locations, in a statistical sense. In practice, each sample typically represents hundreds or thousands of locations.

For the basic SHARDS algorithm, we simply take this spatial sampling filter, and place it in front of a standard reuse-distance algorithm, effectively scaling down its inputs by a factor of R. We then take the reuse distances output by the algorithm, and scale them back up, to reflect the sampling rate R.

This method has several desirable properties. As required for reuse distance computations, it ensures that all accesses to the same location will be sampled, since they will have the same hash value. It does not require any prior knowledge about the system, its workload, or the location address space. In particular, no information is needed about the set of locations that may be accessed by the workload, nor the distribution of accesses to these locations. As a result, SHARDS sampling is effectively stateless. In contrast, explicitly preselecting a random subset of locations may require significant storage, especially if the location address space is large. Often, only a small fraction of this space is accessed by the workload, making such preselection especially inefficient.

Although this basic approach can reduce the time and space required to generate an MRC by several orders of magnitude, it can still be improved. First, the required space grows slowly, but isn't bounded, making it hard to use in memory-constrained environments. Second, choosing an appropriate sampling rate can be challenging, since it implies an accuracy versus overhead tradeoff that can be difficult to evaluate, especially in an online system.

To address these issues, we developed a fixed-size version of SHARDS that operates in constant space. The basic idea is that instead of specifying the sampling rate R, we specify a maximum number of samples to track, s_{max}. Placing a hard bound on the sample set results in a constant-space algorithm. The basic spatial filtering step operates exactly the same as before. But now, if adding a new sample would exceed the space bound s_{max}, some existing sample must be evicted to make room.

We remove the sample with the maximum hash value, T_{max}, closest to T. The global threshold T is then lowered to T_{max} *since* any larger values cannot fit in the set, reducing the sampling rate R dynamically. When the threshold is lowered, a *subset-inclusion property* is maintained automatically; each location sampled *after* lowering the rate would also have been sampled *prior* to lowering the rate.

The subset-inclusion property is leveraged to lower the sampling rate adaptively as more unique locations are encountered, in order to maintain a fixed bound on the total number of samples that are tracked at any given point in time. The sampling rate is initialized to a high value; in practice $R_0 = 0.1$ is sufficiently high to achieve good results with nearly any workload.

As the rate is reduced, the counts associated with earlier updates to the reuse-distance histogram need to be adjusted. Ideally, the effects of all updates associated with an evicted sample should be rescaled exactly. Since this would incur significant space and processing costs, we opt for a simple approximation.

When the threshold is reduced, the count associated with each histogram bucket is scaled by the ratio of the new and old sampling rates, R_{new} / R_{old}, which is equivalent to the ratio of the new and old thresholds, T_{new} / T_{old}. Rescaling makes the simplifying assumption that previous references to an evicted sample contributed equally to all existing buckets—a reasonable statistical approximation when viewed over many sample evictions and rescaling operations. Rescaling is performed incrementally and inexpensively, and ensures that subsequent references to the remaining samples have the appropriate relative weight associated with their corresponding histogram bucket increments.

Evaluating SHARDS

With a constant memory footprint, SHARDS is suitable for online use in memory-constrained systems, such as device drivers in embedded systems. To explore such applications, we developed a high-performance implementation, written in C, and optimized for space efficiency. With our default setting of s_{max} = 8K, the entire measured runtime footprint—including code size, stack space, and all other memory usage—is smaller than 1 MB, making this implementation practical even for extremely memory-constrained execution environments.

We have deployed SHARDS in the context of the commercial CloudPhysics I/O caching analytics service for virtualized environments. Our system streams compressed block I/O traces for VMware virtual disks from customer datacenters to a cloud-based backend that constructs approximate MRCs efficiently. A Web-based interface reports expected cache benefits, such as the cache size required to reduce average I/O latency by specified amounts. Running this service, we have accumulated a large number of production traces from customer environments.

We analyzed 106 week-long traces, collected from virtual disks in production customer environments with sizes ranging from 8 GB to 34 TB, with a median of 90 GB. The associated virtual machines were a mix of Windows and Linux, with up to 64 GB RAM (6 GB median) and up to 32 virtual CPUs (2 vCPUs median). In addition, we used 18 publicly available block I/O traces from the SNIA IOTTA repository [6], including a dozen week-long enterprise server traces collected by Microsoft Research Cambridge [3].

In total, we analyzed a diverse set of 124 real-world block I/O traces to evaluate the accuracy and performance of SHARDS compared to exact methods. For each experiment, we modeled a simple LRU cache replacement policy, with a 16 KB cache block size—typical for storage cache configurations in commercial virtualized systems.

To quantify the accuracy of SHARDS, we considered the difference between each approximate MRC, constructed using spatially hashed sampling, and its corresponding exact MRC, generated from a complete reference trace. An intuitive measure of this distance, also used to quantify error in related work, is the mean absolute difference or error (MAE) between the approximate and exact MRCs across several different cache sizes. This difference is between two values in the range [0, 1], so an absolute error of 0.01 represents 1% of that range.

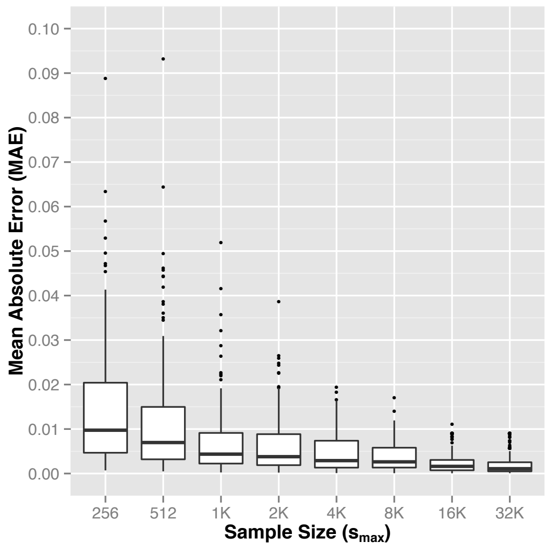


Figure 3: Error analysis. Mean absolute error calculated over all 124 traces for different SHARDS sample set sizes. The top and bottom of each box represents the first and third quartile values of the error; the thick black line is the median. The thin whiskers represent the min and max error, excluding outliers, which are represented by dots.

The box plots in Figure 3 show the MAE metric for a wide range of SHARDS sample set sizes (s_{max}). For each trace, this distance is computed over all discrete cache sizes, at 64 MB granularity, corresponding to all non-zero histogram buckets. Overall, the average error is extremely low. For s_{max} = 8K, the median MAE is 0.0027, with a worst case of 0.017. The error for tiny sample sizes is also surprisingly small. For example, with only 256 samples, the error for 75% of the traces is below 0.02, although there are many outliers.

Many statistical methods exhibit sampling error inversely proportional to $\sqrt{n}$, where n is the sample size. Our data is consistent; regressing the average absolute error for each s_{max} value shown in Figure 3 against $1/\sqrt{s_{max}}$ resulted in a high correlation coefficient of $r^2 = 0.98$. This explains the observed diminishing accuracy improvements with increasing s_{max}.

Why does SHARDS work so well, even with small sample sizes and correspondingly low sampling rates? Our intuition is that most workloads are composed of a fairly small number of basic underlying processes, each of which operates somewhat uniformly over relatively large amounts of data. As a result, a small number of representative samples is sufficient to model the main underlying processes. Additional samples are needed to properly capture the relative weights of these processes. Interestingly, the number of samples required to obtain accurate results for a given

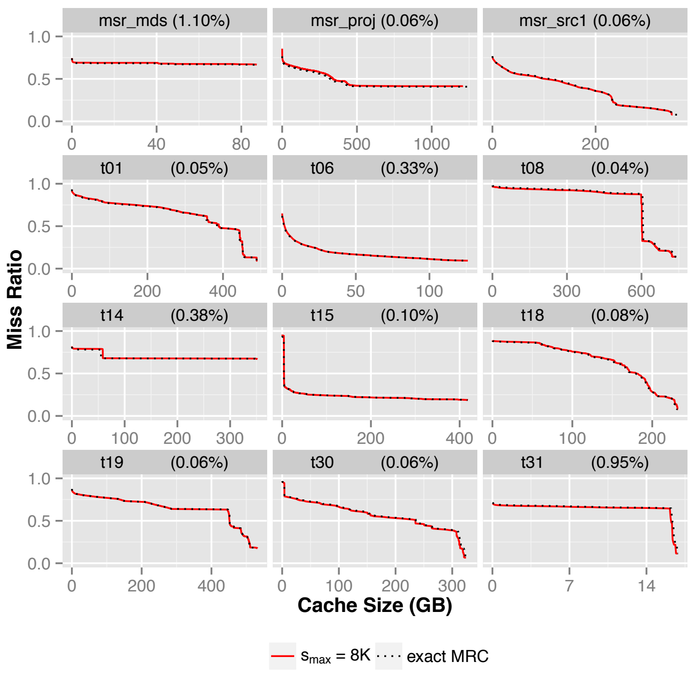


Figure 4: Example MRCs: exact vs. SHARDS. Exact and approximate MRCs for 12 representative traces. Approximate MRCs are constructed using SHARDS with s_{max} = 8K. Trace names are shown for three public MSR traces [3]; others are anonymized. The effective sampling rates appear in parentheses.

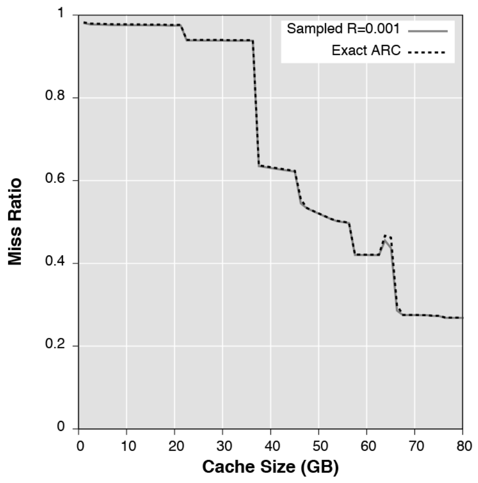


Figure 5: Scaled-down ARC simulation. Exact and approximate MRCs for the MSR-web disk trace [3]. Each curve plots 64 separate ARC simulations at different cache sizes.

workload may be indicative of its underlying dimensionality or intrinsic complexity.

Figure 4 provides further qualitative evidence of SHARDS accuracy for a dozen representative traces. In most cases, the approximate and exact MRCs are nearly indistinguishable. Each plot is annotated with the effective dynamic sampling rate, indicating the fraction of I/Os processed, including evicted samples. This rate reflects the amount of processing required to construct the MRC.

Overall, quantitative experiments confirm that, for all workloads, SHARDS yields accurate MRCs, in radically less time and space than conventional exact algorithms. Compared to the sequential implementation of PARDA [4], a modern high-performance reuse-distance algorithm, SHARDS requires dramatically less memory and processing resources. For our trace set, we measured memory reductions by a factor of up to 10,800χ for large traces, and a median of 185χ across all traces. The computation cost was also reduced up to 204χ for large traces, with a median of 22χ. For large traces, SHARDS throughput exceeds 17 million references per second.

Renewed Interest in MRCs

Recently, there has been renewed interest in algorithms for efficient MRC construction, using a variety of different techniques, which has been very exciting to see. For example, Saemundsson et al. [5] grouped references into variable-sized buckets. Their ROUNDER aging algorithm with 128 buckets yields MAEs up to 0.04 with a median MAE of 0.006 for partial MRCs, but the space complexity remains $O(M)$.

Wires et al. recently created an alternate way of computing MRCs using a *counter stack* [8]. In the closest matching test case using the same large trace and an identical cache configuration, Counter Stacks is more than 7χ slower and needs 62χ as much memory as SHARDS with s_{max} = 8K. In this case, Counter Stacks is more accurate, with an MAE of only 0.0025, compared to 0.0061 for SHARDS. Using s_{max} = 32K, with a 2 MB memory footprint, SHARDS yields a comparable MAE of 0.0026, still approximately 7χ faster, with a 40χ smaller footprint. While Counter Stacks uses $O(\log M)$ space, SHARDS computes MRCs in small *constant* space. As a result, it is practical to use separate, potentially concurrent SHARDS instances to efficiently compute multiple MRCs tracking different properties or time-scales for a given reference stream.

Scaled-Down Simulation

Like other algorithms based on Mattson's single-pass method [1], SHARDS constructs MRCs for caches that use a stack-algorithm replacement policy, such as LRU. Significantly, the same underlying spatial sampling approach can be used to simulate more sophisticated policies, such as ARC [2], for which there are no known single-pass methods to speed up analysis.

Our approach is to simulate each cache size separately, while scaling down the simulations to regain efficiency. As with basic SHARDS, input references are filtered using a hash-based sampling condition, corresponding to the sampling rate R. A series of separate simulations is run, each using a different cache size, which is also scaled down by R. Figure 5 presents both exact and scaled-down sampled MRCs for the public MSR web block trace [3], for 64 simulated ARC cache sizes. With R = 0.001, the simulated cache is only 0.1% of the desired cache size, achieving huge reductions in space and time, while exhibiting excellent accuracy, with an MAE of 0.002.

Encouraged by our results from generalizing hash-based spatial sampling to model sophisticated cache replacement policies, we are exploring similar techniques for other complex systems. We are also examining the rich temporal dynamics of MRCs at different time scales.

References

[1] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger, "Evaluation Techniques for Storage Hierarchies," *IBM Syst. J.,* vol. 9, no. 2 (June 1970), 78–117.

[2] N. Megiddo and D. S. Modha, "ARC: A Self-Tuning, Low Overhead Replacement Cache," in *Proceedings of the 2nd USENIX Conference on File and Storage Technologies (FAST '03),* Berkeley, CA, 2003, USENIX Association, pp. 115–130.

[3] D. Narayanan, A. Donnelly, and A. Rowstron, "Write Off-Loading: Practical Power Management for Enterprise Storage," *Trans. Storage,* vol. 4, no. 3 (Nov. 2008), 10:1–10:23.

[4] Q. Niu, J. Dinan, Q. Lu, and P. Sadayappan, "PARDA: A Fast Parallel Reuse Distance Analysis Algorithm," in *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium (IPDPS '12),* Washington, DC, 2012, IEEE Computer Society, pp. 1284–1294.

[5] T. Saemundsson, H. Bjornsson, G. Chockler, and Y. Vigfusson, "Dynamic Performance Profiling of Cloud Caches," in *Proceedings of the ACM Symposium on Cloud Computing (SOCC '14),* New York, NY, 2014, ACM, pp. 28:1–28:14.

[6] SNIA. SNIA IOTTA Repository Block I/O Traces: http://iotta.snia.org/tracetypes/3.

[7] C. A. Waldspurger, N. Park, A. Garthwaite, and I. Ahmad, "Efficient MRC Construction with SHARDS," in *13th USENIX Conference on File and Storage Technologies (FAST '15),* Santa Clara, CA, 2015, USENIX Association, pp. 95–110.

[8] J. Wires, S. Ingram, Z. Drudi, N. J. A. Harvey, and A. Warfield, "Characterizing Storage Workloads with Counter Stacks," in *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI '14),* Berkeley, CA, 2014, USENIX Association, pp. 335–349.

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

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

A campus rep's responsibilities include:

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

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

To qualify as a campus representative, you must:

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

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

usenix
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

SYSADMIN

Daemon Management Under Systemd

ZBIGNIEW JĘDRZEJEWSKI-SZMEK AND JÓHANN B. GUÐMUNDSSON

Zbigniew Jędrzejewski-Szmek works in a mixed experimental-computational neuroscience lab and writes stochastic simulators and programs for the analysis of experimental data. In his free time he works on systemd and the Fedora Linux distribution. zbyszek@in.waw.pl

Jóhann B. Guðmundsson, Penguin Farmer, IT Fireman, Archer, Enduro Rider, Viking-Reenactor, and general insignificant being in an insignificant world, living in the middle of the North Atlantic on an erupting rock on top of the world who has done a thing or two in distributions and open source. johannbg@gmail.com

The systemd project is the basic user-space building block used to construct a modern Linux OS. The main daemon, systemd, is the first process started by the kernel, and it brings up the system and acts as a service manager. This article shows how to start a daemon under systemd, describes the supervision and management capabilities that systemd provides, and shows how they can be applied to turn a simple application into a robust and secure daemon. It is a common misconception that systemd is somehow limited to desktop distributions. This is hardly true; similarly to the Linux kernel, systemd supports and is used on servers and desktops, but it is also in the cloud and extends all the way down to embedded devices. In general it tries to be as portable as the kernel. It is now the default on new installations in Debian, Ubuntu, Fedora/RHEL/CentOS, OpenSUSE/SUSE, Arch, Tizen, and various derivatives.

Systemd refers both to the system manager and to the project as a whole. In addition to systemd, the project provides a device manager (systemd-udevd), a logging framework (systemd-journald), and a daemon to keep track of user sessions (systemd-logind). For server and VM environments, reliability, control over daemons, and uniform management are possibly the most important focus points, and for desktop, emphasis is placed on a whole-system view of users, secure access to hardware, and quick boot times. The latter is also important for VMs, containers, and embedded devices. Watchdog integration, factory reset, and read-only root are useful for containers and embedded devices. Systemd also has very strong integration with LSMs, with SELinux and AppArmor support coming mainly from the server and container users, and SMACK (Simplified Mandatory Access Control Kernel) used in smartphones and embedded devices.

Uniformity is good for sysadmins and developers alike, and the systemd project has become the de facto standard base of Linux systems. After the last release of systemd, packages for Fedora, Debian, and Ubuntu appeared on the same day. This creates synergy and allows developers from all distributions to participate directly in upstream development, which in turn has led to a renewed focus on bug fixes and new features. Relying on the presence of systemd in all major distributions and across the stack makes it possible to make full use of functions that systemd provides. This article will strive to show how simpler and more secure daemons can be created.

Systemd Units

Systemd keeps the state of the system in a graph of interconnected "units." The most important are "services," i.e., daemons and scripts, with each service unit composed of one or more processes. Other unit types encapsulate resources ("device," "mount," "swap," "socket," "slice" units), group other units ("target," "snapshot," "scope"), or trigger other units under certain conditions ("timer," "path," and "socket" units). Units *may* be configured through files on disk, but this is not necessary for all unit types. Device units are dynamically created based on the device tree exported by the kernel and managed by udev. Similarly, mount and swap units are usually generated from /etc/fstab but are also dynamically based on the current set of mounted file systems and used swap devices.

Unit files on disk have an extension that corresponds to the type of the unit (e.g., httpd.service, sshd.socket). Unit files are plain-text files in a syntax inspired by the desktop entry specification [1]. Out of the 12 unit types, this article only deals with two (.service and .socket) and only a few configuration options. The full set of configuration directives in unit files is rather large (258 as of systemd release 219), so the reader is referred to the copious documentation [2].

The Basics of Systemd Services

Systemd manages daemons and other jobs as "services." Each service is described by a simple declarative text file that lists the commands to execute. The service unit file also contains a short description of the service, pointers to documentation, and a list of dependencies on other services.

Service files have the .service extension and are usually located in /usr/lib/systemd/system, when they are installed as part of a package, or in /etc/systemd/system, when they are local configuration.

systemctl is the tool used to manipulate and check unit status.

This article will use a simple Python server as an example, and you can download the script and other files [3] and follow along. To keep things simple, but not totally trivial, the server will provide a hashing service and will respond with cryptographic hashes of the data it is sent.

A unit file for this server could be:

```
# /etc/systemd/system/hasher.service
[Unit]
Description=Text hashing service
Documentation=https://example.com/hasher
[Service]
ExecStart=/usr/bin/python -m hasher
```

That's it—after copying this file to one of the directories listed above, the daemon can be started with

```
systemctl start hasher
```

and stopped with

```
systemctl stop hasher
```

Wrapping daemons in initialization scripts that parse options and prepare state is discouraged. Systemd service configuration is intentionally not a programming language, and the logic of service initialization and state transitions is embedded in the boot manager (systemd) itself. Preferably, the daemon should be able to be launched directly. For the cases where this is not possible, it *is* possible to invoke arbitrary shell commands by either embedding them directly as calls to /bin/sh -c '...' in the unit file or by executing an external script.

/usr, /etc, /run Hierarchies and .d Snippets

Systemd units that are distributed as part of the operating system are installed in /usr/lib/systemd/system/ (/etc/systemd/system/ is reserved for the administrator). If a service with the same name appears in both places, the one in /etc is used. This allows the administrator to override distro configuration. /run/systemd/system serves a similar purpose and can be used to add temporary overrides, with a priority higher than /usr but lower than /etc.

Adding a unit file with the same name completely replaces the existing unit. Very often, just a modification or extension of the original unit is wanted. This is achieved by so called "drop-ins"—configuration files with like syntax that are located in a directory with the same name as the unit, suffixed with ".d," (e.g., hasher.service.d/).

For example, we would like to run our service under Python 3. We are not sure how this will work out, so we create an override in /run. It will be wiped out after reboot. "systemctl edit" can be used to conveniently create drop-in snippets: it will create the directory and launch an editor.

```
$ systemctl edit --runtime hasher
# /run/systemd/system/hasher.service.d/override.conf
[Service]
ExecStart=
ExecStart=/usr/bin/python3 -m hasher
```

If we just added a new ExecStart line, systemd would merge them into a list of things to execute. By specifying an empty ExecStart=, we first clear the previous setting.

After updating unit configuration, the changes are not picked up automatically. This restarts the unit using the updated configuration:

```
$ systemctl daemon-reload
$ systemctl restart hasher
```

"systemctl cat" can be used to print the main service configuration file and all drop-ins.

Socket Activation

Socket activation was one of the early flagship features of systemd, introduced in current form in the original systemd announcement [4]. Motivation included simplification of daemons, simplification of dependencies between them, and uniform configuration of sockets on which daemons listen. Since then, the rest of systemd has grown, but socket activation remains a crucial building block and has become the basis of various security features.

Socket activation, depending on the context, can mean a few things. Let's tackle them one at a time.

The most important part of socket activation is that the daemon does not itself create the socket it will listen on, but it inherits the socket as a file descriptor (file descriptor 3, right after standard input, output, and error). This socket can either be a listening socket, which means that the daemon will have to listen(2) on it and serve incoming connections from clients, or just a single connection, that is, a socket received from accept(2). This latter version is rather inefficient, since a separate process is spawned for each connection, so this article will only describe the first version. In this version, after the daemon has been spawned, there is absolutely no difference in efficiency compared to the situation in which the daemon itself opens the sockets it listens on.

The way in which systemd informs the daemon that the sockets have already been opened for it is by means of two environment variables. $LISTEN_FDS contains the number of sockets. For example, for an httpd server, which listens on both HTTP and HTTPS (ports 80 and 443), those two sockets could be given as file descriptors 3 and 4, and $LISTEN_FDS would be 2. The second environment variable, $LISTEN_PID, sets the process identifier of the daemon. If the daemon spawns children but forgets to unset $LISTEN_FDS, this second variable acts as a safety feature because those children will know that $LISTEN_FDS was not addressed to them.

Systemd provides a library (libsystemd), which contains a utility function [5] to check $LISTEN_PID and query $LISTEN_FDS. sd_listen_fds() will unset those variables so they are not inherited by children. Nevertheless, if libsystemd is not a good fit for any reason, this protocol is so simple that it can be trivially reimplemented in any language that allows file descriptors to be manipulated.

To configure socket activation for a systemd service, a .socket file is used. Continuing with our example, the following would cause systemd to open TCP port 9001 for our daemon:

```
# /etc/systemd/system/hasher.socket
[Unit]
Description=Text hashing service socket
[Socket]
ListenStream=9001
```

By default a .socket unit is used with the .service unit of the same name, so we don't need to name hasher.service explicitly.

Types of Socket Activation

The word "activation" in "socket activation" implies that the connection to the socket causes the daemon to start. This used to be true (under inetd) but is just one possibility under systemd. In general, systemd can be configured to activate services for more than one reason, combining the functionality that was traditionally split between the init scripts, inetd, crond, and even anacron. A service can be configured to always start on boot or to start as a dependency of another service, which corresponds to the traditional "start at boot" semantics. It can also be started as a result of an incoming connection, like inetd would do. It can also be started at a specific time or date, a certain time after boot, after some interval after it last stopped running, which covers the functionality provided by crond and anacron, and a bit more. Some more esoteric triggers, like a file being created in a directory or another daemon failing, can also be used.

Having all this functionality in the system manager has certain advantages. The way that the daemon is started is configured only once. Race conditions between different activation mechanisms are handled gracefully: if a connection comes in while the daemon is still initializing, it will be serviced once the daemon is ready. If a daemon has to be started as a dependency of two different daemons, it will be started just once. If a cron-job-style service that is supposed to be started every day takes a few days to run, it will not be started twice.

Returning to a socket activated daemon, this daemon will inherit its socket or sockets already open. It can be configured to be started at boot, in which case socket activation only means that the sockets are opened before and not after the fork. The daemon can also be configured to start lazily on an incoming connection, in which case this first connection will not be lost, but it will be handled with a delay because the daemon needs to start first. In case of subsequent connections there is no difference in either case. Systemd "units" that describe the service and the socket can be written to allow both modes to be supported and can be enabled with a single command. How to do this will be described in the next section.

Systemd supports IPv4 and IPv6, and TCP and UDP sockets, but it also supports UNIX sockets, FIFOs, POSIX message queues, character devices, /proc and /sys special files, and netlink sockets. All those can be passed to the daemon using the socket activation protocol. Systemd will also configure various TCP/IP socket options, the congestion algorithm and listen queue size, binding to a specific network device, and permissions on UNIX sockets. The author of the daemon still needs to write code to support stream or datagram connections of course. The advantage is that they need not bother with writing code to parse addresses and configure different protocols and families. The advantage for the administrator is that all that can be uniformly configured and enabled with simple declarative switches in the unit file, if the daemon provides necessary support.

So far our daemon was listening on the wildcard address (::). Let's say we would like to have it listen on the specific address 10.1.2.3 instead. The kernel will not allow binding to an address before it has been configured. To avoid having to synchronize with network initialization, we can use IP_FREEBIND, controlled by unit option FreeBind.

The updated unit file looks like this:

```
# /etc/systemd/system/hasher.socket
...
[Socket]
ListenDatagram=10.1.2.3:9001
FreeBind=yes
```

Enabling Units to Start by Default

The previous section mentioned that systemd services can be configured to start based on a few different conditions, including starting "at boot." Systemd groups services, which are described by .service units, into targets, described by .target units, for easier management. During boot, systemd starts a single target, including all of its dependencies. So starting a service during boot simply means adding it to the right target. For normal services this is multi-user.target, which contains everything that is part of a normally running system.

Back to our example, since our service does not work unless systemd hands it an open socket, we add a dependency on the .socket unit. We also specify how the service should be enabled. We append to the .service file:

```
# /etc/systemd/system/hasher.service
[Unit]
Description=Text hashing service
Requires=hasher.socket
[Service]
...
[Install]
WantedBy=multi-user.target
```

which allows the administrator to enable and disable the service. Let's do that:

```
  $ systemctl enable hasher.service
  Created symlink from /etc/systemd/system/multi-user.target.
wants/hasher.service to /etc/systemd/system/hasher.service.
```

This symlink encodes the dependency. Creating a symlink from a .wants/ directory is an alternative to specifying Wants=hasher in the multi-user.target file that does not require modifying the unit file. Specifying all dependencies a unit needs to successfully start is the principle underlying service management by systemd. In this specific case, systemd tries to start multi-user .target, which depends on hasher.service, which depends on hasher.socket, so the socket will be started first, then the service, and in the end systemd will announce that it has reached the specified target.

Socket units are described by .socket files. They too can be configured to be created at boot by adding them to a target. For sockets this is sockets.target, which itself is part of multi-user. target. Similarly to .service units, we add installation instructions to the socket unit:

```
# /etc/systemd/system/hasher.socket
...
[Install]
WantedBy=sockets.target
```

When the socket+service pair is written this way, the administrator can enable just the socket to have lazy activation on the first connection, or can enable the service to always start it during boot.

Let's make our daemon lazily activated:

```
  $ systemctl disable hasher.service
  Removed symlink /etc/systemd/system/multi-user.target.wants/
hasher.service.
  $ systemctl enable hasher.socket
  Created symlink from /etc/systemd/system/sockets.target.
wants/hasher.socket to /etc/systemd/system/hasher.socket.
```

Using Socket Activation to Resolve Dependencies

Sockets.target is actually started early during boot. This means that systemd opens the sockets even before the services that will handle the connections can be started. Traditionally, the administrator had to make sure that services are started in the right order, so that they manage to open their sockets before the services which will try to connect to those sockets are started. Socket activated services can be started in parallel, even if they are interdependent. A service that connects to a socket belonging to a daemon that hasn't started yet will simply wait. As long as no loops exist, those dependencies are resolved without any explicit configuration.

Incidentally, opening sockets in this fashion ensures that they are not accidentally opened by a different program, solving the problem that portreserve deals with.

Failure Handling

There are two schools of thought on how to handle crashing daemons. The first states that daemons should not crash and must be fixed. The second states that crashing daemons are a fact of life and have to be dealt with. Systemd watches the status of all services and will notice if the main process of a service exits for any reason. The manager can be configured to take some action. Restart=on-failure, a common setting that causes a service to be restarted when it exits with a non-zero exit code, is killed by SIGSEGV, SIGABRT, or similar, or if a timeout is hit.

To monitor service health, systemd supports basic watchdog functionality. When WatchdogSec= is specified in the service file, systemd provides the $WATCHDOG_USEC variable in the environment of the service and expects periodic notifications with the sd_notify() call. When the service does not send the heartbeat signal, it will be aborted, and possibly restarted, depending on the settings described above.

The watcher is also watched. The manager can be configured to enable the hardware watchdog (with RuntimeWatchdogSec= and ShutdownWatchdogSec= settings) to allow the system to be automatically restarted if PID 1 stops responding. This way a chain of supervision from the hardware to the leaf services is established.

The Journal and Log Labeling

One of the most debated aspects of systemd is its log handling. Systemd-journald is one of the non-optional parts of systemd and is one of the first services to start and one of the last services to be stopped. All messages are stored in a binary format in /var/log/journal/<machine-id> and can be read using journalctl or using sd_journal_get_data(3) and related functions found in libsystemd. Even traditional syslog daemons nowadays usually do this, and get their data from binary journal files before writing them to text files. There are reasons for this organization.

Systemd tries very hard not to lose any log messages. All messages from services will be captured and processed by systemd-journald. This includes syslog messages and anything written to standard output and standard error. Daemons that crash, especially during startup, will often print the cause to standard error, so it is nice to capture those messages too. In addition, structured messages can be sent to the journal through a custom UNIX socket. sd_journal_print(), and related functions in libsystemd provide this functionality. The advantage is that in addition to the main message, additional fields can be attached. Each entry in the journal is composed of a group of FIELD=VALUE pairs. The “message” is stored as MESSAGE field, and other fields can carry text or binary content. In fact, sd_journal_print() will by default attach the source code filename and line.

The binary format which systemd-journald uses indexes messages based on field names and field values. Taking our hashing daemon as an example, it could add the remote address and port as additional fields when using native journal logging. The journal can then be queried for messages about a certain remote from certain dates without searching through all logs:

```
 journal.send('New connection on fd={} from {}:{}'.format(fd,
address, port),
             ADDRESS=address[0],
             PORT=port)
```

Traditional syslog messages include an identifier, usually the program name. In journal messages this field is called SYSLOG _IDENTIFIER and is controlled by the sender.

```
systemctl -t <identifier>
```

can be used to query messages with a certain identifier.

Another important aspect of structured logs is that journald attaches some fields that specify the provenance of the message and cannot be faked or modified by the sender. Those fields are labeled with a leading underscore (e.g., _PID, _UID, _GID, _SYSTEMD_UNIT for the process identifier, user and group of the sender, and the systemd service containing the process). Traditional syslog does not have anything like this and allows any sender to send messages on behalf of any daemon. Systemd makes extensive use of those additional fields. When starting a service or logging anything related to a service, messages about the service are tagged with the name of the service. In addition, privileged daemons like setroubleshoot will also tag messages as pertaining to a certain service. Systemd-journald also reads audit messages using the netlink socket and parses them to extract process identifiers and other metadata. This allows all messages *from* a service and *about* a service to be retrieved with a simple

```
journalctl -u <service>
```

command. This same functionality is used when showing service status:

```
systemctl status <service>
```

will list the processes being part of the service, and what systemd knows about the service, but also the last ten lines of logs pertaining to the service.

If we were to start hasher.service

```
$ systemctl start hasher
```

we could see messages from systemd and from the daemon interwoven:

```
 $ journalctl -u hasher
 Feb 22 19:15:12 fedora systemd[1]: Starting Text hashing
service...
 Feb 22 19:15:12 fedora python3[222]: /usr/bin/python3: No
module named hasher
 Feb 22 19:15:12 fedora systemd[1]: hasher.service: main
process exited, code=exited, status=1/FAILURE
 Feb 22 19:15:12 fedora systemd[1]: Unit hasher.service
entered failed state.
```

How does journalctl know which messages to show? If we also show the auxiliary data, we can see that messages are tagged with either _PID=1 and UNIT=hasher.service or _SYSTEMD_UNIT=hasher.service. The first comes from PID 1, and journalctl knows that it can trust the UNIT= field. The second is tagged as coming from the service itself.

Systemd-journald is started very early. In fact, if systemd is used in the initramfs, systemd-journald runs in the initramfs, writing logs to temporary storage under /run/log/. After the transition to the main file system, systemd-journald continues writing to /run/log, and then flushes those logs to /var/log/ after /var/ has been mounted. This means that logs from early boot are available just like the rest.

Systemd-journald watches the amount of free disk space and will not allow journal files to eat up all available space. In the default configuration it will cap the total space used and also leave a certain percentage of the disk free.

Security Features

A very simple yet effective way to limit the damage that a hacked or misbehaving service can do is to run it under its own user. The primary reason for socket activation is the simplification of network daemons and their lazy activation. But it has implications for security, too. If a daemon does not open sockets by itself, it can be less privileged. A daemon that wants to listen on a port below 1024 can be started under root, open the port itself, and then do the fairly complicated transition to unprivileged user itself. But if systemd opens the port for it, it can run as the unprivileged user from the start.

Letting systemd take care of the user transition is trivial: use the set User= option.

Our process could still transition back by running a SUID binary. We can disallow this and any other transitions or privilege escalation with NoNewPrivileges=yes.

Some daemons need to run as root, but systemd can still restrict them by using mount and network namespaces which limit their view of the world. A group of settings use mount namespaces [6] to curtail access to the file system. ProtectHome= and ProtectSystem= are the high-level options. The first can be used to present /home to the daemon as either empty or read-only, and the second will make /usr, /boot, and optionally /etc read-only for the daemon. ReadOnlyDirectories=, ReadWriteDirectories=, InaccessibleDirectories= are the low level settings that do what their names suggest.

Using predictable file names in shared temporary directories is a common source of denial-of-service and security vulnerabilities. PrivateTmp= setting uses mount namespaces to give private /tmp and /var/tmp directories to the daemon. This protects both the daemon from users and other daemons, and others from the daemon.

Systemd uses network namespaces to prevent a daemon from using the network. A service running with PrivateNetwork=yes sees only a private loopback device. If the daemon is compromised, it cannot be used to exfiltrate data or attack other hosts.

Paradoxically, socket-activated network daemons are often started with PrivateNetwork=yes. This means that they can be run locked down as an unprivileged user, and their only means of contact with the network is through the sockets inherited from systemd.

Let’s turn those additional protections on for our service:

```
  # /etc/systemd/system/hasher.service
  …
  [Service]
  User=hasher           ← the primary group of the user is used
too if Group= is not specified
  NoNewPrivileges=yes
  ProtectHome=yes
  ProtectSystem=full    ← "full" includes /etc in addition to /
usr and /boot
  PrivateTmp=yes
  PrivateNetwork=yes
```

After the service is restarted, it has its own network namespace with a private lo device, cannot see /home or write to /usr, /boot, and /etc even if the file access mode would allow. What the service sees as /tmp is really a directory /tmp/systemd-private-<bootid>-hasher.service-<gibberish>/tmp.

SELinux and Other Linux Security Modules

There are two parts to the integration with a security module: the first part is that systemd will perform initial SELinux configuration when the system is brought up. Systemd is aware of SELinux contexts, so when creating files or opening sockets systemd will label them properly. The second part is the ability to override default domain transitions with configuration in unit files. The SELinuxContext= setting can be used to set the context of executed processes. Similar support and settings exist for AppArmor and SMACK. Integrating support directly in the boot manager means that initialization is performed very early, thus LSM protection includes the early boot, and any initialization errors are caught and will be treated as fatal if necessary. It should be noted that SELinuxContext is logged as a trusted metadata in the journal.

Resource Limits

Traditional UNIX resource limits were applied per process (niceness, virtual memory size, CPU usage, open files) or per user (number of processes). Those limits are still supported and can be set automatically with Nice=, LimitDATA=, LimitCPU=, LimitNOFILE=, LimitNPROC=, and others [7].

Each systemd service runs in its own control group. Cgroups allow resource limits to be applied at the level of the whole service or group of services, and to partition resources more fairly. By default, systemd only uses the cgroup hierarchy to keep track of forked children of various services. Additional settings can be used to turn on specific controllers and constrain resource usage. It should be noted that this is not free, and especially the memory controller is known for its high overhead. Systemd presents a simplified subset of the functionality provided by the kernel, and can limit and partition CPU usage (CPUShares=), memory usage (MemoryLimit=), and block device bandwidth (BlockIOWeight=, BlockIOReadBandwidth=, BlockIOWriteBandwidth=).

Automated Management

Systemd binaries generally produce two kinds of output on the console: colorful tabularized output for human consumption, and plain output useful for scripting. This second type is stable [8] and can be used as a basis for management tools. The most popular tools like Chef, Puppet, and Salt all provide similar functionality that wraps calls to systemctl enable/disable/start/stop/restart/is-active and can be used to manage units in a centralized manner.

Systemctl is actually a wrapper around the D-Bus API of systemd and defers operations on units to it. The status of all units is also available over D-Bus, and D-Bus property notifications can be used for live updates. The kcmsystemd control module for KDE uses this, and hopefully more tools will in the future.

Summary

The systemd stack has grown over the last few years, to the point where it is simply impossible to describe more than some aspects in a short article like this. We show how the facilities provided by systemd can be used to build a secure daemon by making use of user separation, namespaces, control groups, and resource limits. Those futures are provided by the Linux kernel but are not as widely used as they should be because of the additional work required to support them. Our daemon (although written in Python) is also fairly efficient and robust: it uses epoll, provides extensive logs, and will be monitored and restarted if necessary. We hope that this proves that systemd makes the lives of developers and programmers more pleasant.

Resources

[1] http://standards.freedesktop.org/desktop-entry-spec/latest/.

[2] http://www.freedesktop.org/software/systemd/man/systemd.directives.html#Unit%20directives.

[3] Python script and configuration examples: http://in.waw.pl/git/login-article-example/.

[4] The original systemd announcement: http://0pointer.de/blog/projects/systemd.html.

[5] sd_listen_fds() in libsystemd: http://www.freedesktop.org/software/systemd/man/sd_listen_fds.html.

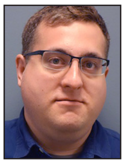
[6] James Bottomley and Pavel Emelyanov, "Containers," *;login:*, vol. 39, no. 5, October 2014: https://www.usenix.org/publications/login/october-2014-vol-39-no-5/containers.

[7] CPU limits for systemd services: http://www.freedesktop.org/software/systemd/man/systemd.exec.html#LimitCPU=.

[8] Systemd interface stability: http://www.freedesktop.org/wiki/Software/systemd/InterfacePortabilityAndStabilityChart/.

Hiring Site Reliability Engineers

CHRIS JONES, TODD UNDERWOOD, AND SHYLAJA NUKALA



A computational daemonologist, Chris Jones works in San Francisco as a Site Reliability Engineer for Google App Engine, a platform serving over 28 billion requests per day. He was previously responsible for the care and feeding of advertising statistics, data warehousing, and customer support systems, joining Google in 2007. In other lives, Chris has worked in academic IT, analyzed data for political campaigns, and engaged in some light BSD kernel hacking, picking up degrees in computer engineering, economics, and technology policy along the way. cdjones@google.com

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

Shylaja Nukala is a Technical Writer at Google, and she has been leading the Site Reliability Engineering (SRE) technical writing team for six years. She is involved in documentation, communication, and training for SRE. Prior to Google, she worked at Epiphany and Sony. She has a PhD from the School of Communication and Information, Rutgers University. She also taught at Rutgers, Santa Clara, and San Jose State universities. snukala@google.com

Operating distributed systems at scale requires an unusual set of skills—problem solving, programming, system design, networking, and OS internals—which are difficult to find in one person. At Google, we've found some ways to hire Site Reliability Engineers, blending both software and systems skills to help keep a high standard for new SREs across our many teams and sites, including standardizing the format of our interviews and the unusual practice of making hiring decisions by committee. Adopting similar practices can help your SRE or DevOps team grow by consistently hiring excellent coworkers.

Google's Site Reliability Engineering (SRE) organization is a mix of software engineers (known as SWEs) and systems engineers (known as SEs) with a flair for building and operating reliable complex software systems at an incredible scale. SREs have a wide range of backgrounds—from a traditional CS degree or self-taught sysadmin to academic biochemists; we've found that a candidate's educational background and work experience are less predictive than their performance in interviews with future colleagues. Google's hiring process intentionally prevents teams' managers from making hiring decisions, instead using a hiring committee of engineers from across the organization to assess the merits of each potential hire on a case-by-case basis.

Who We Look For

Ben Treynor, Google Vice President and Site Reliability Tsar, describes SRE as being "what you get when you treat operations as if it's a software problem": a software engineering philosophy ("write software to solve problems") hybridized with an operations mission ("keep the service running"). These two influences can be seen in the dual job titles within SRE—SRE-Systems Engineer and SRE-Software Engineer—reflecting the different emphasis with which individual SREs may approach the same problems: one may be most comfortable writing new software, while the other may tend to prefer fitting existing components together into new and exciting architectures, but everyone can do some of both.

By "systems engineering," we mean a discipline that takes a holistic approach to the *connections* between distinct software systems or services rather than either (1) the internals of how to build a piece of software, where software engineering has tended to concentrate as a field, or (2) how software artifacts are deployed onto specific hardware, which has been the historic domain of system administration. Instead, systems engineers view the collection of individual pieces, which may be built by many separate product development teams, as a whole with properties distinct from its components. SEs tend to focus on how to monitor services, identify and remove bottlenecks, manage and balance connections, handle data replication, ensure data resiliency, and so on. This skill becomes essential at the scale at which Google and other large software organizations operate.

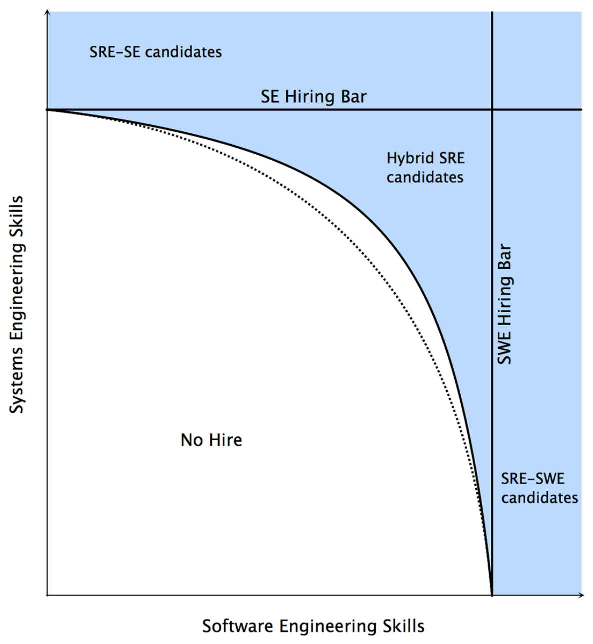


Figure 1: Skills and hiring for SRE candidates

Google's preference for generalists and internal mobility meshes well with our hiring bar for SWE candidates within SRE being the same as that for our product development organizations—engineers are able to move freely from the reliability organization to other groups. We've found, however, that there's a third category of candidates who form a particularly valuable pool: those who trade off some *depth* of experience in one field for a *breadth* of experience in both fields. In Figure 1, the area in the shaded region between the curve with acceptable tradeoffs (which we call the "Treynor Curve"), the SE hiring bar, and the SWE hiring bar shows this pool of "hybrid" SRE candidates.

What We Look For

We look for candidates who are smart, passionate about building and running some of the largest and most complex software artifacts on the planet, and able to quickly understand how something works that they may never have seen before. Since we would like to scale the size of our systems *much* faster than Google can hire SREs to work on them, the SRE approach to problem resolution emphasizes automation, improving system design, and building resilience into our systems so that we don't have to repeatedly fix the same problems; it's also much more interesting to find new failure modes, usually due to newly launched systems or features. Accordingly, we try to find and hire candidates wherever they are, regardless of background: there are simply too few people with the right mindset and skills for SRE to limit ourselves to candidates with conventional backgrounds.

Every SRE, regardless of whether they're an SE or SWE, needs to have an understanding of the fundamentals of computing. Unsurprisingly, we look for the ability to solve problems with software, whether that's been acquired from a textbook or at the school of hard knocks. Similarly, troubleshooting skills and the ability to unpack a problem into smaller pieces, identify possible causes, triage, and do so systematically are essential, whether that's been acquired through debugging code, operating a network, building hardware, or in other, entirely unrelated domains; the cognitive skills and approaches to problem-solving are subject-matter agnostic and critical to have, regardless of a candidate's background.

We specifically do not look for "architects"—that's a role that simply doesn't exist at Google: everyone in our engineering organizations both designs and implements. Similarly, prospective candidates for managerial roles in Site Reliability must meet the same technical bar as individual contributors, as well as understand that management at Google is a different proposition from that at many other companies: SRE line managers are typically also technical contributors to their team, including being part of an on-call rotation, in addition to their managerial responsibilities in coordinating skilled and highly autonomous individual contributors.

How We Interview

SRE interviews follow Google's typical engineering interview pattern: much like elsewhere in the industry, there is first a short technical pre-screen with a recruiter; next, an initial phone interview with an engineer, perhaps with a follow-up phone interview; and then a day at one of our sites, doing four or five interviews, each with an engineer. Each interview is intended to be a conversation between peers rather than an interrogation: we strenuously discourage brainteasers and trivia questions, as they provide minimal insight into how a candidate thinks about problems.

Each SRE interviewer has a specified topic to cover—e.g., programming, UNIX internals, networks, or troubleshooting and problem-solving—to ensure that we have a wide range of assessments from interviewers, while minimizing duplication [1]. The mix of topics varies based on the candidate's self-assessed strengths and weaknesses: there's no point in spending valuable interview time asking someone about their weaknesses, only to discover that they were right when they said they didn't know much about a topic. Similarly, we try to match candidates' strengths with those of their interviewers, so that they have a more interesting conversation and there's a better quality signal in the resulting assessment. Ideally, each interviewer will discover the limits of the candidate's knowledge in their topic and see how the candidate reasons and reacts when faced with problems they have not previously encountered—that is, can they make reasonable assumptions and extrapolations from what they *do* know?

At least one interview will involve programming in the candidate's preferred language: while Google uses C++, Go, Java, Javascript, and Python for most of its projects, we have SREs who can read pretty much any language a candidate might want to use. Candidates do not need to use one of the five languages in their interview, as we expect that anyone who meets our hiring bar is likely able to learn at least one of those languages fairly quickly.

One of the interviews will be on "non-abstract large system design" [2], in which they're asked to concretely design a large-scale system, such as a system to join different types of log entries written in multiple datacenters for analysis. Simply laying out boxes on a whiteboard and invoking magic technologies ("I'll store everything in BigTable, since that's what Google uses") to solve a problem isn't sufficient: silver bullets are rarely found when building real software systems, so it would leave too much mystery about a candidate's quality to accept answers depending on them. Instead, we're looking for candidates to be able to provide realistic estimates of throughput, storage, and so on for each component—while considering various tradeoffs for reliability, cost, and difficulty of building the system. The ideal candidate can not only reason about how each high-level component fits together, but work through each layer in the design, right down to the hardware underpinning it.

Afterwards, the interviewer provides a hire/no-hire recommendation along with detailed written feedback explaining how a candidate answered the questions and the strengths or weaknesses of those responses compared to others'.

If the interview feedback for a candidate is borderline, the recruiter can ask a group of engineers to perform some quality control and validation: Are additional interviews likely to be needed? Is the interview panel sufficiently senior for the candidate's experience? Does another topic need to be covered or an interview topic repeated for some reason?

How We Decide

An unusual feature of Google's engineering hiring process is that the hire/no-hire decision is *not* made by a manager; instead, it's made by a hiring committee before going to senior management for approval (if the decision was to hire). The committee members are drawn from across the organization, including multiple locations and teams within SRE.

A mix of SRE managers and individual contributors will read the interview feedback each interviewer wrote and come to a joint decision about whether the hiring bar for the role has been met. Hiring committee meetings are characterized by extensive debate on whether each candidate meets our organization-wide hiring bar. The hiring committee has access to past scores and hiring decisions for each interviewer, so it can decide how much weight to put on an interviewer's feedback given their past predictive track record.

Using a committee to make hiring decisions is a critically important part of our process because it ensures that we have an assessment that reflects the skills and capabilities we expect our engineers to have, while maintaining common standards between offices and parts of the organization to ensure internal mobility. The committee's diverse perspectives can also provide a broader assessment of candidate strengths and weaknesses.

Removing the (prospective) hiring manager from the process prevents the common management pathology of taking the first warm body who seems vaguely competent to fill a vacancy or a short-term need, compromising hiring standards at the expense of the long-term health of the organization. In fact, allocation of a new hire to a specific team in SRE always happens separately, *after* the hire/no-hire decision is made. As a result, we can expect our hiring quality across SRE to stay consistent over time—or at least, be changed *intentionally* by management in response to headcount availability—rather than simply choosing a candidate who happened to apply for a particular opening at the discretion of that team's manager. As it happens, Google's practice is to hire candidates we believe to be better than our average current employee [3], consciously accepting a higher risk of false negatives (incorrect no-hire decisions) to reduce the chance of false positives (incorrect decisions to hire).

Conclusion

Talented future SREs are scarce and hard to find; it's often difficult to make a confident prediction about whether a given candidate will succeed as an SRE. We've found that standardizing our hiring process so that we consistently cover a range of skills essential for success in SRE and ensuring that all SRE candidates are able to code regardless of their background in system administration, systems engineering, or software engineering are critical to guaranteeing a high level of mobility between SRE teams and within organizations at Google.

Finding people who are *simultaneously* generalists comfortable with encountering novel software systems and specialists with sufficient technical depth in particular fields (e.g., software engineering, networks, distributed systems) is even more difficult: by building an organization that takes each SRE's individual strengths—regardless of his or her place on the Treynor Curve between systems engineering and software engineering—and combines them, we're able to have an *organization* which can paradoxically bridge the two skills.

On Interviewers

Ideally, we would like every interview to be performed by long-tenured, senior SREs who have done thousands of interviews and have a perfect track record of predicting hiring decisions; unfortunately, the volume of interviewing and other demands on Senior Engineers' time make this an impossibility—and this would also make it impossible for anyone else to become an experienced interviewer. Instead, we try to populate an interview panel with a majority of reasonably experienced interviewers whose feedback has good predictive value, while still providing an opportunity for newer interviewers to get practice in one of the interview slots. Very new interviewers may "shadow" experienced engineers' interviews or "reverse shadow," in which one conducts the interview while the other observes: both submit feedback, but only the experienced interviewer's feedback is used.

As engineers gain experience interviewing, they become better able to determine candidate strength through more exposure to interview candidates and common interview responses, both good and bad; increased time working with their peers to understand the skills expected of new hires; and the opportunity to write assessments and receive feedback from colleagues on those interviews. After some time, we are able to evaluate their hiring recommendations and feedback for interview quality, consistency, and predictive value.

Because phone interviews are a single point of failure—a candidate's rejection at this stage generally precludes further consideration for that role for some time—we choose phone interviewers from a relatively small pool of particularly consistent interviewers trusted by the hiring committee, to try to make sure that we make good decisions about who to invite for on-site interviews. This is intended to ensure that candidates who make it to that stage have a realistic prospect of making it through the interviews and being hired, reducing the cost of interviewing: each on-site candidate costs at least four hours for the interviews themselves, plus time spent on writing feedback and reviewing it in the hiring committee.

We have an organized pool of interview questions with canonical answers we've seen from past candidates for interviewers to draw upon. This makes it easier for newer interviewers to get started and provides a consistent subset of questions for the hiring committee to use in comparing candidates, although interviewers are free to add their own technical questions. Over the course of an interview, an interviewer refines and increases the technical depth of the conversation to determine the candidate's depth of understanding, so that the pool is used more as a starting point for further discussion and elaboration rather than being a list of trivia questions to be checked off in sequence. Each interview is thus unique, though it follows a common pattern.

Several locations with SRE teams have a regular "Interview Club" group, where SREs can try out potential interview questions to see how they work in practice and to get feedback from experienced interviewers. SREs are also encouraged to occasionally observe hiring committee meetings. They may also receive comments from the hiring committee on their interview notes to help make their future feedback more useful or might mention that a particular approach to an interview question worked well.

Practices We've Found Helpful for Hiring SREs

Structure interviews to cover the topics essential to the SRE role, as appropriate for the candidate's skills and strengths; assign a specific topic to each interviewer.

- Build a pool of interview questions along with "gold standard" responses, to provide a consistent subset of questions across candidates.
- Ask about how to build *concrete* large-scale systems; avoid brainteasers and trivia.
- Ask *every* SRE candidate to code *something*.
- Separate interviewing from hiring decisions.
- Make hire/no-hire decisions by a committee of engineers.

On Process

TODD UNDERWOOD

It's reasonable to ask why Google uses such an elaborate process to hire people. Some other companies manage to hire people somewhat or much faster than Google. Perhaps there should be a model of quick hire and, if things aren't working out, quick fire. There are a number of reasons why this cannot work well at Google and probably doesn't work well at most other places, either.

As we pointed out, we hire generalists who will likely be part of several teams over their careers at Google. It's critically important that our hiring standard not be lowered by an individual hiring manager's short-term need for staffing. The easiest way to avoid this temptation while maintaining uniform and high standards is to make the hiring decision through a committee that excludes the hiring manager.

Additionally: the learning curve at Google is quite high. Our software stack is sophisticated, fragile, complex, and powerful, and it takes quite a while to learn it. It therefore takes months before it is apparent whether a new hire is doing well. By the time a bad fit is obvious, we may have made an investment of many months. This necessarily encourages us to be much more conservative than some other employers about hiring decisions.

Finally, there's the very serious issue of bias. Hiring decisions made quickly by individuals often result in hiring people who are just like those doing the hiring. The technology industry has a bias problem, and we are committed to doing what we can to fix it. Some of the things we have learned about avoiding bias in decisions, especially where that bias is unconscious, is that making decisions as a group according to articulated standards helps. It also helps to justify the decision and know that you'll have to justify the decision in advance. By incorporating these aspects into our process, we hope to make decisions that add diversity.

Hiring decisions made quickly by a hiring manager according to no articulated standards might work well at some organizations, but we have come to believe that consistently hiring well is critically important to us, and employment is critically important to most of our employees. Taking an appropriate amount of time to make sure there is a reasonable fit makes good sense for us.

Acknowledgments

This article was based on a conversation between Gary Arneson, Zoltan Egyed, Chris Jones, and Todd Underwood. Thanks to Dermot Duffy, David Hixson, Alex Matey, Niall Murphy, and Lucas Pereira for their invaluable comments.

Resources

[1] See Daniel Kahneman, "Thinking, Fast and Slow" (2011) for further discussion of using a structured hiring format instead of making intuitive judgments; also Lazlo Bock's *Work Rules*: https://www/workrules.net.

[2] Google SREs often teach classes on non-abstract large system design at LISA and other venues, featuring exercises where small groups solve design problems much like those encountered by interview candidates.

[3] Peter Norvig: googleresearch.blogspot.com/2006/03/hiring-lake-wobegon-strategy.html.

The Systems Engineering Side of Site Reliability Engineering

DAVID HIXSON AND BETSY BEYER

David Hixson is a Technical Project Manager in the Site Reliability organization at Google, where he has been for eight years. He currently spends his time predicting how social products at Google will grow and trying to make the reality better than the plan. He previously worked as a system administrator on High Availability systems and has an MBA from Arizona State. usenix@dhixson.com

Betsy Beyer is a Technical Writer specializing in virtualization software for Google SRE in NYC. She has previously provided documentation for Google Data Center and Hardware Operations teams. Before moving to New York, Betsy was a lecturer in technical writing at Stanford University. She holds degrees from Stanford and Tulane. bbeyer@google.com

In order to run the company's numerous services as efficiently and reliably as possible, Google's Site Reliability Engineering (SRE) organization leverages the expertise of two main disciplines: Software Engineering and Systems Engineering. The roles of Software Engineer (SWE) and Systems Engineer (SE) lie at the two poles of the SRE continuum of skills and interests. While Site Reliability Engineers tend to be assigned to one of these two buckets, there is much overlap between the two job roles, and the knowledge exchange between the two job roles is rather fluid.

The collaborative SWE/SE engineering approach was popularized in the Silicon Valley environment, but is now common to locations characterized by a high density of software engineering requiring an operational component. A hybridization of the two skill sets is also demonstrated by lone individuals who hold together complicated software systems by sheer force of will. While Software Engineering is generally well understood in the tech world, Systems Engineering remains a bit more nebulous. What exactly is Systems Engineering at one of these companies? This article takes a closer look at Systems Engineers: what they are, what they do, and how you might become one.

Characteristics of a Systems Engineer

The task of defining the exact characteristics of a Systems Engineer at Google, or at any other Silicon Valley tech company that uses classifications like "Developer Operations," is problematic. Informally, a Systems Engineer might be described as someone who enjoys discovering particularly difficult problems and applying their problem-solving skills in uncharted territory. A Systems Engineer regularly undertakes tasks like dismantling software or hardware, re-engineering and optimizing their design, or finding new uses for the components. Traditional wisdom dictates that "you know a Systems Engineer when you see one." However, this definition fails to either permit a Systems Engineer to self-identify or to become a better Systems Engineer. Nor does it help tech companies to create a satisfying job ladder and career progression for Systems Engineers.

Therefore, pinning down a set of characteristics particular to a Systems Engineer is a necessary and useful exercise.

A Systems Engineer

...uses the scientific principles of experimentation and observation to build a body of knowledge that affects the architecture and design of the system as a whole.

Complex and ambiguous problems can be solved by:

1. Breaking down the problem into smaller components.
2. Testing assumptions about these components.
3. Continuing in this vein of investigation until a root cause is identified.

In addition to this troubleshooting skill set, a Systems Engineer must have the willingness to chase a problem through the multiple layers beyond its surface, acquiring the knowledge necessary to conduct the investigation along the way. Willingness to learn new technologies or techniques is critical to pursuing each new investigation.

...has an actionable skepticism towards layers of abstraction.

At least a few layers of abstraction are necessary in order to deal with the complexity in the world around us. However, when a Systems Engineer's expectations of how a certain system should perform are violated, the engineer must figure out why. The investigative effort can focus on determining the causes of existing problems, avoiding future problems, or finding improvements where no one has looked before.

...focuses on the connections between the components within the system as much as focusing on the components themselves.

Understanding the interactions between each system element is critical to building or troubleshooting systems that scale. Decisions about how communications are passed between the elements can have extreme effects on the overall stability of the system.

...knows many ways to *not* solve a problem, rather than one perfect way to solve the problem.

A perfect solution to a problem is extremely rare. Instead, choosing the best engineering solution requires tradeoffs between many different elements. Deliberately evaluating and making these tradeoffs is key to building a stable and scalable system or to identifying problems in an existing system. To state this principle another way: success is probably a corner case of the possible failure modes of a complex system.

Differences Between Software Engineering, System Administration, and Systems Engineering

The fundamental differences among three core specializations in creating and operating software at tech companies—Software Engineering, System Administration, and Systems Engineering—fall into three main categories: approaches to problems, academic background and professional communities, and career progression.

Approaches to Problems

The scenario presented in the simple drawing to the right (Figure 1) would be approached in very distinct manners by an archetypical SWE, SA, and SE. In practice, a successful Site Reliability Engineer is expected to use a combination of these approaches.

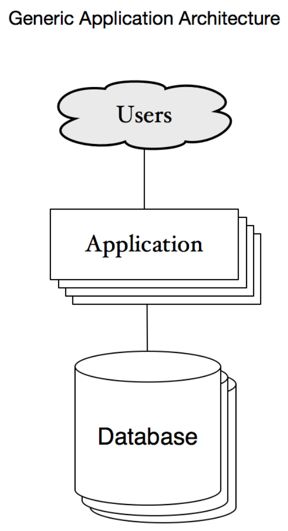


Figure 1: Generic application architecture

An SWE would focus on constructing boxes that have predictable behavior and that operate as efficiently as possible. Software needs to:

- Turn requests into responses.
- Call the database via the appropriate API.
- Optimize the schema to reflect the kinds of queries that will be made.

A combination of language choices, frameworks, unit tests, load testing, and a variety of best practices make these operations more likely to work effectively.

Sysadmins generally approach their operational responsibilities with operational solutions. This diagram would prompt an SA to think about the infrastructure required to actualize the service represented and manage the operational aspects, considering questions such as:

- What hardware is needed?
- What software is implied but not specified?
- How do we get all of these operations to run at the same time in a supportable way?

A wide variety of tasks not depicted in the drawing come into scope: OS deployment, backups, security, user administration, configuration management, logs, monitoring, performance tuning, capacity planning, and so forth.

Systems Engineers generally approach operational requirements with a software or system approach. This diagram would prompt an SE to focus on the lines connecting the sets of boxes and the overall experience represented in the diagram more than on the boxes themselves. For example, an SE might ask the following questions:

- If multiple front ends are employed, how do we shard incoming traffic?
- How do we replicate databases, and how do we manage connections, failover, hotspots, etc.?
- Do we replicate databases globally, in the same datacenter, or on the same machines?
- What changes are we sensitive to in terms of impacting availability or performance?

An SE's scope encompasses everything from language choice to networking to hardware platforms. However, the SE considers the code in less depth than would an SWE, and the machines in less depth than would an SA.

Academic Background and Professional Communities

Software Engineering is taught through a variety of academic paths, ranging from elementary school courses to doctoral programs. The academic community has produced a substantial body of respected work, and research into various facets of computing continues to advance the state of the art. Ongoing advancements in computer science, performed by a thriving community spanning multiple industries, continually facilitate the authoring of better code both through technology and best practices.

System Administrators usually receive some level of professional training, which normally doesn't occur through traditional academic institutions, or at least not at the level of a degree program. Much of an SA's training and certification is on the job and focuses on specific hardware or software configurations from specific vendors. Therefore, an SA's training risks becoming highly specialized and may be outdated fairly quickly. A large body of documentation focuses on the practice of the SA job (e.g., best practices for accomplishing specific tasks), but there is little writing that explains the philosophy behind the job. SA support communities exist, but are heavily fragmented because SA work is very specialized and concerns quite specific focus areas, such as backups, user management, storage, and so on.

Systems Engineering is more multi-disciplinary than Software Engineering or System Administration, meaning that it benefits heavily from academic study, but doesn't necessarily align with a typical degree program that focuses on depth rather than breadth. SEs benefit from increasing their skills in computer science or other areas of engineering, but these fields don't represent the whole of an SE's work. Academic disciplines dubbed "Systems Engineering" do exist, but typically don't focus on the kinds of work that information technology companies expect. There are communities which overlap with the situations faced by SEs, but the discussions of such communities tend to be far-ranging and cover the union of several different complex areas. As a result, the audience that can usefully sympathize or contribute to solutions tends to be rather limited.

Career Progression

SWE positions exist at every level in a job ladder, from those able to code "hello world" to the engineers in charge of inventing the technologies of the future. Similarly, SWE positions exist across an incredible variety of available technologies and scope—an SWE might focus on writing custom firmware for some exotic device, designing a new programming language, building Skynet, or writing an iOS app that issues reminders to buy groceries.

The SA ladder often starts with a help desk position or work servicing computers, advances to managing or tuning complex services, and further advances to managing networks of computers. Advancement is generally either one of scale—extending up to controlling thousands of computers—or depth—requiring expertise in managing a smaller number of much more complex systems.

The SE suffers from not having a low ladder rung from which to ascend. A productive SE must have the skills and experience that stem from working on real problems. Subsequently, an SE likely begins on either an SWE or SA ladder, at some point recognizing more of a cross-functional calling and branching off into an SE role. A typical SWE to SE trajectory entails either working on loosely coupled systems, or needing to tune a software project for a very specific role that crosses into computer hardware or networking territory. A typical SA to SE trajectory entails either fitting together a wide variety of components that operate outside of SA documentation, or needing to understand and modify software in interesting ways.

At the apex of the Silicon Valley job ladders, the SE and SWE jobs merge back together, since the largest and most complicated software problems cross so many disciplines and technologies that the leaders of such projects need to understand how all of their components fit together. These engineers are dubbed Software Engineers or Principal Engineers, but no element of a given system is outside of their scope—everything from software, to networking, to hardware choices are within their purview. Because pushing the frontiers of technology is rarely conducted in just one dimension, the tradeoffs between these choices need to be conducted with as much flexibility and understanding as possible. SRE at Google works to push this merger of skills early in engineers' careers in order to provide them with opportunities to impact or create globally distributed systems.

The SE Approach to Problems

The manner in which an SE approaches a problem varies from person to person and depends on the service being investigated, but a few SE-specific skills and thought processes are quite common. Problem-solving begins with figuring out how a given system is supposed to work and/or the expected outcome of the system. An SE's investigative approach starts at the unexpected output or outcome and then traverses the system until the problem is resolved. At each step in the investigation, the SE contemplates the expected event or outcome versus what actually occurs. When the expected and the actual don't align, the SE digs deeper.

An SE doesn't just investigate correctness, as problems frequently split off into much less obvious areas such as how to avoid latency tails, the construction of systems that are resilient to failure conditions, or the design of systems that can run under extremely high levels of load. At each stage of investigation, it is critical to not just understand what should happen, but to create tests to verify that a given event occurs in the expected manner and with no side effects worthy of consideration. Any unexpected event is an opportunity to deconstruct the component, be it software or hardware, and repeat the process in order to understand the event at a lower level of abstraction.

The skills espoused by an SE can be vital in bringing an idea to fruition in a way that is scalable, performant, and generally in alignment with the expectations of all players involved. Unfortunately, success in the areas of SE expertise is frequently difficult to detect, particularly by those who aren't deeply involved in the investigative process and resulting decisions. At the end of the day, success is rarely attributed to the integration between system components (an operation performed by the SE), but rather to whomever built its key components (the SWE). Although Google SRE works hard to correct this anti-pattern by raising the level of understanding and appreciation of SE tasks, Systems Engineering is a good career choice for those seeking the satisfaction afforded by intensive problem solving, but is not an attractive role for those seeking recognition outside their immediate team.

The deliverable of an SE is unlikely to be a body of code comparable to that of a software engineer. Instead, the job of an SE entails the thought process and work necessary to either make a given system function as intended, or to build elegant solutions to new problems. The important contribution of an SE is the improvement to the system as a whole, not the list of actions, and quality and effectiveness must be measured by the improvement of an entire system over time, rather than the delivery of a particular, narrow task. The deliverable may be just a few lines of code, or even a setting on some obscure piece of networking hardware that ends up providing value [1]. Subsequently, evaluating an SE's overall performance or contribution to any given project is difficult. While the SE role is potentially higher impact than that of an SWE or SA, it is a difficult role to manage, and career advancement paths aren't always obvious.

Becoming a (Better) SE

To pursue work (or improvement) as an SE, start by building enough depth in one area to provide a basis from which to understand how the pieces of your system interact. You can begin such an investigation from either an SA or an SWE background. Note that if you're pursuing an SE role at the launch of your career, a basis in software engineering may be advantageous, as the available education options are both more comprehensive and more readily available. That being said, building your SE skill set requires refusing to acknowledge that a system problem lies outside of your scope or control. Follow networking problems over the network, chase performance problems out of your code and into the hardware, or dissect an application to figure out how it works and then improve the application. Whenever a system violates your expectations of how it should perform, figure out why. Working with or contributing to open source projects is one great way to improve your SE-related skills. Systems Engineers tend to love open source, as it enables them to pry open a black box and shine a light upon its inner workings to figure out exactly why components behave the way they do. Growing your software engineering skills will help with career advancement, as these skills enable a deeper engagement with the software components that comprise the systems with which SEs work.

Fundamentally, skill as a Systems Engineer comes from satisfying your curiosity over and over again, and accumulating that experience to continually improve your investigative skills. Eventually, this accumulated experience can inform how you build new systems. From day one, incorporate the system monitoring intended for your final product. Design systems that expose their side effects in a way that makes those effects easy to understand later. Document expected outcomes and your assumptions about any given system, in addition to what might be expected if you violate those assumptions.

For those who possess an abundance of curiosity and a willingness to constantly dig into uncertainty and complexity, Systems Engineering can be a thoroughly enjoyable and rewarding career path. The SE role has matured into a viable profession, and the prospects for SEs will likely continue to grow in the future. More and more companies, particularly those outside the traditional tech world, will need Systems Engineers to build and improve the complex systems required to sustain their operations.

Resource

[1] For more context on SE deliverables, see snopes.com: Know Where Man: http://www.snopes.com/business/genius/where.asp.

/var/log/manager
Surreal Management Situations

ANDY SEELY

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

The IT manager's day is filled with the mundane. Time cards. Shift schedules. Metrics reports. Counseling employees. Writing evaluations. Signing things. Meetings. Meetings. Meetings. And then there are rare moments of novelty, when something strange happens and you say, "Ahh, that wasn't what I expected at all." It's rarely something both strange and good, but even strange and bad can be better than the monotony of much of the manager's function.

Welcome to the Team. You're Fired

There was an employee on another contract task who was losing his position when the task was completed. He had a reputation for doing good work, and I could use someone with his skills in setting up and maintaining audio-visual systems. I tried to make a place for him in my staffing plan, but I wasn't able to move fast enough and he aged out of the company through a standard layoff. This meant that once I finally had the position lined up, I had to rehire him as an external candidate rather than as an internal transfer. Negotiate. Plan. Line up. Click "hire now" in the Human Resources system. And then I turned to the next thing on my list and figured I'd see him on his first day the following week.

The next day I was contacted by the recruiter. Then by the HR support person. Then by the onboarding person. And then by a random person who didn't even have a stake in the hiring action. Each told a similar story, how difficult and angry and rude the candidate was. Within hours, it seemed that everyone at my location knew he was a harsh personality, and I was answering people left and right on why I was bringing such a bad egg onto the team.

I talked to him on the telephone and uncovered a sad story. He needed the job. Wanted the job. But was so insulted that he had to reapply as an outsider that he couldn't control his temper and was just lashing out at everyone he had to deal with in the process, including now me on the call. The more he talked, the more he started swearing at me, like he had done with the others in the hiring process. The fabric of the organization was already buzzing with the story and there was no way I could successfully integrate him with the team with all the bad blood.

I had already clicked "hire now," so technically he was an employee who just hadn't signed in on the agreed first day of employment. Welcome to the team. I'm sorry, but you're fired.

The Hundred-Dollar T-Shirt

I have an employee who is impeccably dressed. Whenever I see him, I remember the time in 1989 when I bought a beaten-down 1974 Ford pickup truck for $300. I think of that truck because my employee's shoes cost $300. An exceptional sense of style, with materials, textures, and colors working together seamlessly and in tune with the seasons. And it's not all style over substance: he's my top cybersecurity engineer, and he always gives extra hours and great focus on his task.

There is a manager a rung above me on the organizational chart. This manager is “into” sports. Sports talk frequently takes up more than work talk in any given day. He dresses OK, but sometimes has blue-socks-black-pants problems, and has an unfortunate tendency to wear brown wool slacks with a maroon acrylic polo shirt with a sports logo. It’s a look that doesn’t say “senior manager.”

This manager, dressed in his mismatched sports-logo polo, ran into my cybersecurity engineer in the hallway. My engineer was wearing a sharp ensemble and looked very professional, except that his shirt was lacking a collar. You could call it a “t-shirt” if you wanted to overlook the fact that it probably cost more than $100 at a designer store.

Of course, I had to go and “answer” for my employee later, but I’ll admit I took a certain sad, ironic pleasure in watching the “dress code violation” counseling that the mismatched sports-fan manager felt the need to deliver to my engineer that day.

The Part Is on Order and Will Be Here Any Day

I’ve been pushing a team for months to realign a complex business intelligence system to gain better throughput. My challenge is that the team isn’t technically “mine,” but rather is a peer’s team who is working on something where I’m in more of a customer advocate role. I really feel like I’m more the conscience of the system that keeps reminding the team that they’re not quite done with the job until they do that last bit of performance tuning.

Finally, they tell me they can’t go any further because they need to order a specialized network card for the blade enclosure. Once they have that, they’ll be able to finish re-provisioning LUNs, and then they’ll be able to migrate the virtual hosts into the new configuration. It will all fall into place, just waiting on the part, boss. Don’t worry, boss, we got it, boss.

A month passes and I go back into fact-finding mode, asking if we’re done yet. I get told the same story. We’re waiting on the part, boss. Don’t worry, boss, we got it, boss. I ask, who, exactly, is supposed to be buying the part? It’s our procurements team. Who in procurements? This guy. We call the procurements guy and discovered something fascinating. A few months before the part was ordered, we had upgraded our workflow tracking system. The user interface changed and the procurements guy never went to the training. So we discovered that not only had our part not been ordered, so too had many other things not been ordered because the procurements guy wasn’t ever seeing the service request tickets. We called him and he set up the purchase order on the spot.

Now my “we got it, boss” guy is standing in front of my desk, feeling really badly about all this. I guess we didn’t have it, boss. OK. I ask him what he thought the next potential bottleneck would be. He replied that he’d have the team ready to go as soon as the part arrived, no sir, they’re not going to be a bottleneck. He’ll have them on hot standby to jump to work. They’ll overnight the part, we’ll get started no later than day after tomorrow.

While he was talking, I speaker-phone dialed my buddy at the vendor that sells these special network cards. Hey buddy, I say when he answers, how long to ship one of these special cards? Oh, about three weeks, he answers. I ask him if he can pull a favor, give us a demo or a loaner early, which he agrees to sort out for me. We hang up and I do my honest, level best to not lecture about bottlenecks, responsibilities, and follow-through.

And then I spent the rest of the day feeling good about this particular mess, because I got to be operationally useful instead of just managerially present.

The Preparatory Meeting for the Pre-Meeting for the Meeting

One of my responsibilities is the project management team. One of the final tasks in a project is to conduct an “operational readiness review,” which is the final milestone to have a project accepted by the operations group before we close a project. This is the meeting where we formally hand over diagrams, documentation, systems, and responsibility.

A year ago we had a readiness review that went sour. The project manager wasn’t really prepared. The operations guy had a headache and was testy. Our local NFL team, the Tampa Bay Buccaneers, had just lost to a high school team. There was something in the air that day. And the review meeting went south and was one Molotov cocktail away from a brawl. Ever since then, we’ve held a pre-operational readiness review meeting.

The pre-meeting includes everyone invited to the regular meeting except the actual decision-maker from the operations group. The concept is that if we’re going to fight, let’s get the fight out of us before we talk to the decision-maker. The pre-meeting is where questions get asked and answered so that we’ll know what the answers will be when they’re asked again at the real meeting, and no one is surprised. It feels wasteful to me, but all the major stakeholders seem content, and work isn’t slowing down because of it, so OK, two meetings. One is a dress rehearsal.

I’m in the pre-meeting recently, sitting in the back and observing the relationships, the expertise, the flow, and generally doing my quiet manager thing. And a hockey game breaks out. People swinging sticks, punching, gouging, arguing. Not literally, of course, but there was a strong disagreement over how prepared some people were to have the meeting. Not the real meeting, of course, but the pre-meeting. Not prepared for the pre-meeting. Which was there specifically to help people prepare. Prepare for the real meeting.

A serious manager would have put a halt to what came next, but I was too enthralled to say anything as everyone agreed that from now on they would have a preparatory meeting before the pre-meeting so that everyone would know how to prepare for the meeting that would review the state of everyone's preparedness before they went to the actual meeting.

Maybe We're Just Working Too Hard

These surreal moments are not the norm for the team, although I suspect that these types of things happen in any large organization anywhere. I have a highly educated, experienced, dedicated group of top-flight professionals, but sometimes we all take ourselves a little too seriously, get worked up over the wrong thing, or hold others accountable for the something silly simply as a result of our drive, momentum, passion, and fatigue. The trick is to keep the teams running smoothly for the majority of the time, and to have enough of a sense of humor to poke fun at ourselves when we let things get weird. I'm the manager, and that's my job.

Thanks to Our USENIX and LISA SIG Supporters

USENIX Patrons
Facebook Google Microsoft Research NetApp VMware

USENIX Benefactors
Hewlett-Packard *Linux Pro Magazine* Symantec

USENIX and LISA SIG Partners
Booking.com Cambridge Computer
Can Stock Photo Fotosearch Google

USENIX Partners
Cisco-Meraki EMC Huawei

usenix
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

Donate Today: The USENIX Annual Fund

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

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

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

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

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

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

www.usenix.org/annual-fund

User Groups, Conferences, and Workshops

PETER H. SALUS

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

peter@pedant.com

User groups go back to the beginning of "mass"-produced computers in 1953–54. In this column, I sketch that history, starting with mainframe groups that shared code and tips, up to the founding (and naming) of USENIX and its conferences and workshops.

The first commercial computer, the IBM 701, wasn't completed until late in 1952. The first production machine was shipped from Poughkeepsie to IBM headquarters in Manhattan that December.

Prior to the 701, all computers—Aiken's, Wilkes', ENIAC, etc.—had been one-offs; each was *sui generis*. The 701 was a genuine breakthrough, and IBM built 18 of them! (By way of comparison, Apple announced that it had sold a total of 19 million Macs over fiscal 2014.) On May 7, 1954, the redesigned 701 was announced as the IBM 704.

It was more than merely a redesign.

The 704 was incompatible with the 701. It had 4096 words of magnetic-core memory. It had three index registers. It employed the full, 36-bit word (as opposed to the 701's 18-bit words). It had floating-point arithmetic. It could perform 40,000 instructions per second. While deliveries began in late 1955, the operators (today we would think of them as system administrators) of the eighteen 701s were already fretful months earlier.

IBM itself had no solution to the problem. Although it had hosted a "training class" for customers of the 701 in August 1952, there were no courses, no textbooks, no user manuals. But several of the participants in the training class decided to continue to meet informally and discuss mutual problems.

The participants agreed to hold a second meeting after their own 701s had been installed. That second meeting was hosted by Douglas Aircraft in Santa Monica in August 1953. There were other informal meetings and then, following an IBM Symposium, the RAND Corporation hosted a meeting in Los Angeles in August 1955 of representatives from all 17 organizations that had ordered 704s. It was at this meeting that the world's first computer user group was formed. It was called SHARE.

IBM encouraged the operators to meet, to discuss their problems, and to share their solutions to those problems. IBM funded the meetings as well as making a library of 300 computer programs available to members. SHARE, over 60 years later, is still the place where IBM customers gain information. (A number of the earliest contributed programs are still available.)

The importance of SHARE can be seen in the fact that in December 1955, early purchasers of Remington Rand's ERA-1103A formed an organization called USE (Univac Scientific Exchange). In 1956, user groups for Burroughs and Bendix computers were formed, as well as IBM's GUIDE, for users of their business computers. DECUS (DEC Users' Society) was founded in 1961.

Why should you care? Because today user groups are a vital part of computing's fabric, and it's hard for me to imagine the industry developing without them. But note one thing: each group was hardware-centered and each was funded by a manufacturer.

UNIX Users

When Ken Thompson and Dennis Ritchie gave the first paper on the UNIX operating system in October 1973 at the ACM Symposium on Operating Systems Principles, there was no hardware manufacturer involved. In fact, it was only because an AT&T corporate lawyer interpreted the “consent decree’s” requirement that patents be revealed to include the experimental OS that Ken and Dennis were allowed to drive up the Hudson Valley to IBM’s new research lab at Yorktown Heights and speak to the nearly 200 attendees.

The immediate result was a (small) flurry of requests for the system.

Back to the lawyers.

It was decided that requests from academic institutions were OK. Within six months, Ken had cut and shipped two dozen DEC RK-05s. The RK-05 disk drive was a removable hard drive with a total capacity of 1.6 million 12-bit words. The drive was priced at $5,100; “disk cartridges” cost about $99 each. The software went out with the proviso: “As is, no support!” Or:

> “no advertising
> no support
> no bug fixes”

Later,

> “payment in advance”

was added. Shipped meant mail or UPS. In 1974 there were under 50 hosts on the ARPANET. Only in 1976 did that number reach 63.

Lou Katz, then at Columbia University, was one of the first to request the system. Lou didn’t have an RK-05 drive, so Ken cut a nine-track tape for him. Lou was also the one who realized that if AT&T wasn’t going to support the system, the only answer was for the users to band together. Ken Thompson, complicitously, provided Lou with the list of those who had requested disks. Lou and Mel Ferentz (then at Brooklyn College) sent out invitations, and about 20 people from a dozen institutions turned up for a meeting on May 15, 1974. Thompson was the sole scheduled speaker. DEC problems and tips and UNIX problems and tips were the rest of the agenda.

Twice as many folks attended the meeting of June 18, 1975. It was held at the City University of New York with Ira Fuchs making the arrangements. In October of that year, there were two meetings: on Monday the 27th at CUNY and on Friday the 31st at the Naval Postgraduate School (NPG) in Monterey, CA. Six months later, Bob Fabry organized a two-day meeting at UC Berkeley (February 27–28, 1976)—UNIX user meetings had gone bicoastal.

Lew Law of the Harvard Science Center ran the next two meetings in April and October. The October 1–3, 1976 sessions were the first to top 100 attendees. Five years later, the January 21–23, 1981 meeting organized by Tom Ferrin at UC San Francisco was the first to top 1000 participants.

UNIX News

The June 1975 meeting also gave rise to a purple-dittoed publication, *UNIX NEWS*, the first issue appearing on “June 30, 1975; circulation 37.” It was 11 pages long and contained an institutional mailing list of 33 sites, which revealed how far UNIX had spread in the 18 months since the first presentation: Herriott-Watt University (Scotland), Hebrew University of Jerusalem, Universite Catholique de Louvain (Belgium), and the Universities of Alberta, Saskatchewan, Toronto, and Waterloo (all Canada) were the non-US institutions.

UNIX NEWS Number 2 (October 8, 1975; circulation 60) announced the meeting at the NPG. Number 3 (February 10, 1976) announced meetings at Harvard and UC Berkeley; and Number 4 (March 19, 1976) carried a letter concerning the installation at the University of New South Wales from Dr. John Lions and a new mailing list, with 80 entries.

By September 1976 there were 138. Thirteen were in Canada; ten in the UK; four in Australia; three each in Israel and the Netherlands; and one each in Austria, Belgium, Germany, and Venezuela. The other 101 were in the US.

Unfortunately, the May/June 1977 issue of *UNIX NEWS* was its last. Beginning in July 1977, the publication was called *;login:*. Mel Ferentz had been phoned by an AT&T lawyer and been told that the group (it still had no name) could not use the term UNIX, as they lacked permission from Western Electric. The next meeting on the East Coast was May 24–28, 1978, at Columbia.

At that time, the 350 attendees (!) voted to set up a committee “with the purpose of proposing a set of bylaws for users of UNIX * installations. [* UNIX is a trademark of Bell Laboratories, Inc.]” There were five elected to the committee (Ferentz, Katz, Law, Mars Gralia from Johns Hopkins, and Peter Weiner, founder of Interactive Systems). “Law was elected chairman…the name of the committee shall be the USENIX ** Committee [** USENIX is not a trademark of Bell Laboratories, Inc.].”

The users were already snarky.

Summer and Winter USENIX Conferences

The bicoastal nature of the organization and the academic calendar were the major determinants for the meetings: from 1979 to 2001, conferences were held in January in the West (Santa Monica, 1979; Boulder, 1980; San Francisco, 1981; Santa Monica, 1982) and in June in the East (Toronto, 1979; Newark, DE, 1980). But there was Austin, TX, in June 1981.

USENIX had become increasingly research and academically inclined. As a counterweight, in 1980 a commercial body was formed: /usr/group. In 1983, there was a joint USENIX–/usr/group meeting in San Diego (January 26–28). Over 1800 attended. The next joint meeting was scheduled January 16–20, 1984, in Washington, D.C. Counting government employees, there were 8000 conference goers. And there was a major snowstorm. The result was chaos. (The June 1984 USENIX Conference was held in Salt Lake City, where a more reasonable 1500 attended.)

From 1987–1992, USENIX and /usr/group conferences were held in the same areas but different venues with buses shuttling between them. But this was hampered when January 1987 saw a second Washington, D.C. snowstorm. In August 1989, /usr/group changed its name to UniForum.

Workshops, Too

As installations and users increased, so did interests. In the early 1980s, graphics were a hot topic. In December 1984, USENIX held a "UNIX and Computer Graphics Workshop" in Monterey, CA. The papers were reprinted in the October/November 1985 issue of *;login:*. The same issue carried an announcement/program for a "Second Workshop on Computer Graphics," to be held December 12–13, 1985.

In spring 1986, when I was interviewed for the post of Executive Director of USENIX, the two things the Board focused on were increasing the number of workshops and increasing publications. The first issue of *;login:* (July/August 1986) under my reign of terror included a CFP for the Third Graphics Workshop. The Proceedings of the Second Workshop appeared later that summer.

In January 1987, USENIX issued a CFP for a Fourth Graphics Workshop and one for a "System Administration Workshop." That workshop, chaired by Rob Kolstad and attended by about 50 members, is what grew into LISA. Also in 1987, there was a limited-attendance POSIX workshop, co-chaired by Kirk McKusick and John Quarterman.

USENIX was a hive of activity. It began with a small group at AT&T, but it proliferated through and was implemented by the academic and research user community, a group made up of and run by the users, not a manufacturer.

usenix
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

Publish and Present Your Work at USENIX Conferences

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

Get more details about these Calls at **www.usenix.org/cfp.**

URES '15: 2015 USENIX Release Engineering Summit

November 13, 2015, Washington, D.C.

Submissions due: September 4, 2015

At the third USENIX Release Engineering Summit (URES '15), members of the release engineering community will come together to advance the state of release engineering, discuss its problems and solutions, and provide a forum for communication for members of this quickly growing field. We are excited that this year LISA attendees will be able to drop in on talks so we expect a large audience.

URES '15 is looking for relevant and engaging speakers for our event on November 13, 2015, in Washington, D.C. URES brings together people from all areas of release engineering—release engineers, developers, managers, site reliability engineers and others—to identify and help propose solutions for the most difficult problems in release engineering today.

NSDI '16: 13th USENIX Symposium on Networked Systems Design and Implementation

March 16-18, 2016, Santa Clara, CA

Paper titles and abstracts due: September 17, 2015

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.

FAST '16: 14th USENIX Conference on File and Storage Technologies

February 22-25, 2016, Santa Clara, CA

Submissions due: September 21, 2015

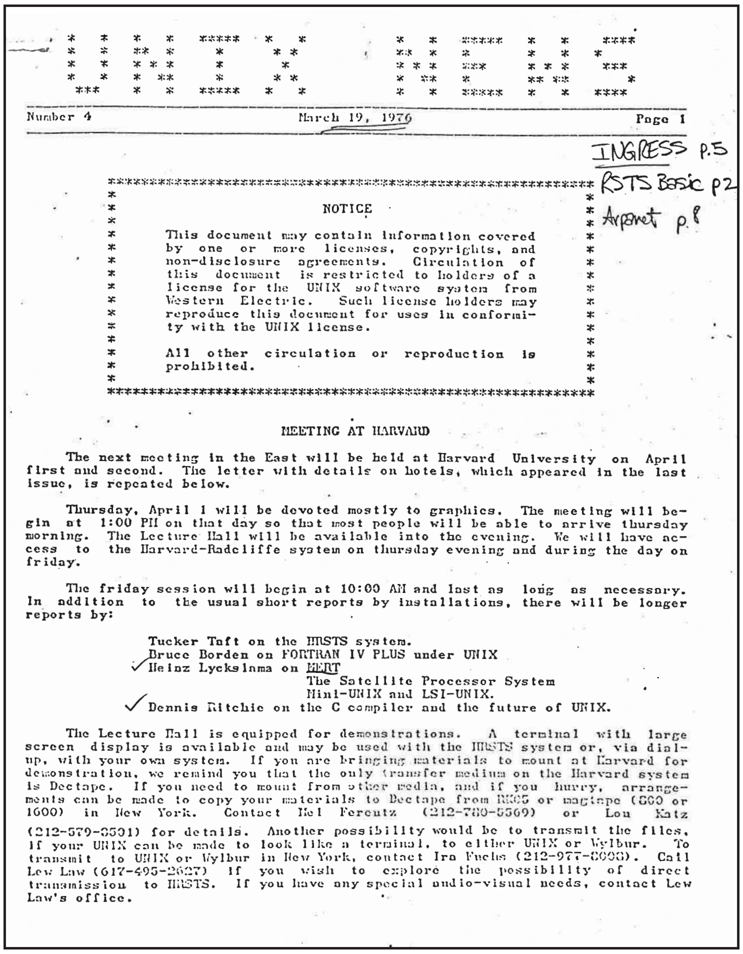
The 14th USENIX Conference on File and Storage Technologies (FAST '16) brings together storage-system researchers and practitioners to explore new directions in the design, implementation, evaluation, and deployment of storage systems. The program committee will interpret "storage systems" broadly; everything 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.

www.usenix.org/cfp

UNIX News

Number 4, March 19, 1976



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

Number 4 March 19, 1976 Page 1

INGRESS p.5
RSTS Basic p2
Arpanet p.8

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

MEETING AT HARVARD

The next meeting in the East will be held at Harvard University on April first and second. The letter with details on hotels, which appeared in the last issue, is repeated below.

Thursday, April 1 will be devoted mostly to graphics. The meeting will begin at 1:00 PM on that day so that most people will be able to arrive thursday morning. The Lecture Hall will be available into the evening. We will have access to the Harvard-Radcliffe system on thursday evening and during the day on friday.

The friday session will begin at 10:00 AM and last as long as necessary. In addition to the usual short reports by installations, there will be longer reports by:

Tucker Taft on the HRSTS system.
Bruce Borden on FORTRAN IV PLUS under UNIX
✓Heinz Lycksinma on MERT
The Satellite Processor System
Mini-UNIX and LSI-UNIX.
✓Dennis Ritchie on the C compiler and the future of UNIX.

The Lecture Hall is equipped for demonstrations. A terminal with large screen display is available and may be used with the HRSTS system or, via dial-up, with your own system. If you are bringing materials to mount at Harvard for demonstration, we remind you that the only transfer medium on the Harvard system is Dectape. If you need to mount from other media, and if you hurry, arrangements can be made to copy your materials to Dectape from RK05 or magtape (800 or 1600) in New York. Contact Mel Ferentz (212-780-5569) or Lou Katz (212-579-3501) for details. Another possibility would be to transmit the files, if your UNIX can be made to look like a terminal, to either UNIX or Wylbur. To transmit to UNIX or Wylbur in New York, contact Ira Fuchs (212-977-8000). Call Lew Law (617-495-2627) if you wish to explore the possibility of direct transmission to HRSTS. If you have any special audio-visual needs, contact Lew Law's office.

UNIX News, Number 4, was published March 19, 1976 by Professor Melvin Ferentz of Brooklyn College of CUNY. We have included excerpts from that issue and have reproduced the text as it appeared in the original, including any typographic errors. Note: We have not included the mailing list and other addresses and telephone numbers that appeared in the original issue, since they are out of date.

Meeting at Harvard

The next meeting in the East will be held at Harvard University on April first and second. The letter with details on hotels, which appeared in the last issue, is repeated below.

Thursday, April 1 will be devoted mostly to graphics. The meeting will begin at 1:00 PM on that day so that most people will be able to arrive thursday morning. The Lecture Hall will be available into the evening. We will have access to the Harvard-Radcliffe system on thursday evening and during the day on friday.

The friday session will begin at 10:00 AM and last as long as necessary. In addition to the usual short reports by installations, there will be longer reports by:

Tucker Taft on the HRSTS system.
Bruce Borden on FORTRAN IV PLUS under UNIX
Heinz Lyckslnma on MERT
The Satellite Processor System
Mini-UNIX and LSI-UNIX.
Dennis Ritchie on the C compiler and the future of UNIX.

The Lecture Hall is equipped for demonstrations. A terminal with large screen display is available and may be used with the HRSTS system or, via dial-up, with your own system. If you are bringing materials to mount at Harvard for demonstration, we remind you that the only transfer medium on the Harvard system is Dectape. If you need to mount from other media, and if you hurry, arrangements can be made to copy your materials to Dectape from RK05 or magtape (800 or 1600) in New York. Contact: Mel Ferentz or Lou Katz for details. Another possibility would be to transmit the files, if your UNIX can be made to look like a terminal, to either UNIX or Wylbur. To transmit to UNIX or Wylbur in New York, contact Ira Fuchs. Call Lew Law if you wish to explore the possibility of direct transmission to HRSTS. If you have any special audio-visual needs, contact Lew Law's office.

Johns Hopkins Software

The Unix Hotline has on it notices from Johns Hopkins University on the availability of several pieces of software.

A version of BASIC-PLUS (version 5B-24) adapted for Unix is available from Johns Hopkins University, Department of Electrical Engineering. It is fully compatible with RSTS/E BASIC-PLUS; nearly any program run on RSTS/E BASIC will work on this one. The only exceptions are:

1) programs are limited to 12K

2) None of the privileged "sys" calls are available; most of the unprivileged ones are.

All of the I/O facilities of BASIC-PLUS (record I/O, virtual arrays, etc.) work on the Unix version: full pathnames can be specified in OPEN, RUN, NAME-AS statements. A "public library" feature allows one to keep frequently used BASIC programs in a directory which is referenced whenever a "$" is prefixed to a pathname.

Binary copies are available to those with a RSTS binary license; source is available to those with a source license. Distribution is available on Dectape or RK disk.

A UNIXed version of MACRO (PDP-11 assembler) is available from JHU Electrical Engineering. It has "macro library" and "cross reference" capability, and is reasonably efficient. To get binary, you need a DEC MACRO binary license for RT11, RSX11D, or DOS; to get source, a DEC source license. Distribution on RK disks or DECtape.

A UNIXed version of Per Brinch-Hansen's "Sequential Pascal" compiler and interpreter are available from JHU-Electrical Engineering The compiler is written in Pascal, consists of 7 passes, and is rather slow. Output from the compiler is run under a separate Pascal interpreter. Compiled files can call the interpreter automatically; hence Pascal programs can be invoked as commands. The lexical conventions have been improved over Brinch-Hansen's original specification; lowercase and tabs are accepted, and [] are used for array specifiers instead of (. .). Most of the UNIX system entries can be invoked from Pascal; users can easily add their own following the design of those already installed.

Prerequisite software: MACRO assembler (availabe from JHU for licensees) and Jeff Rottman's Linker (available from Princeton).

Distribution on RK disk or Dectape. A service charge may be involved.

Mailing List

The attached mailing list is a major revision of the old list. It is based upon a list of licensees dated February 1976 and is ordered on state and by zip code within the state. It is likely that errors have crept in during the editing. Please check your listing and send in corrections. An area of difficulty is multiple installations under a single license. If you know of facilities other than those listed, please let us know.

Software Exchange

As of a few days ago, there were no user submissions to the software exchange. The exchange does have a new C compiler and a new ar.c. We hope the lack of submissions indicates that everybody is busy putting things in shape to send in.

In setting up the exchange, we are hoping that people will send in "trivial" things as well as significant things. The trivial is often the most duplicated in effort.

Manuals

Informal discussions indicate that permission might be granted to the Users' Group to allow a single printer to reproduce the manuals and sell them to us in individual or quantity lots. In order to determine whether this is reasonable or desireable, we need an estimate of the number of copies of each of the manuals you might order. If you are interested in this collective venture please contact Lou Katz (see mailing list).

Notes on West Coast UNIX User's Meeting

Berkeley, California, February 27–28, 1976

From: John Lowry, Carl Sunshine, Steve Zucker

The meeting was attended by about 35 people (half from Berkeley), and hosted by Bob Fabry (Berkeley). (See us for a complete list of participants and addresses.) First there were four major presentations on the UCLA Security Kernel, The INGRES data base system, The Harvard Radcliffe Student Timesharing System (HRSTS), and the Berkeley PDP 11/70 system (Ken Thompson). Then each attendee presented a brief summary of activity at his site. Friday evening and Saturday were devoted to discussing several general topics of interest including interprocess communication, the ARPANET software, multiple-machine UNIX systems, and UNIX development and standards. This note summarizes each of the three areas.

UCLA Security Kernel

Gerry Popek

The object of the UCLA work is to produce a verifiably secure operating system. UCLA has implemented a security kernel in PASCAL (a PASCAL to C translator is available from UCLA) and hope to apply automatic verification to the PASCAL code. The kernel has been designed to be the minimum amount of software that can guarantee data security. The code consists of 4-5K words, including a few drivers, and was initially designed as the basis for a Virtual Machine Monitor (VMM). For the sake of efficiency they have decided to try instead to interface UNIX directly to the security kernel by system calls in the same fashion as MERT, the Bell Labs Multi-Environment Real Time System. They plan to run a separate stripped down version of UNIX in supervisor mode as a part of each UNIX "process" with the various UNIXs communicating by means of shared segments and a kernel supplied message facility. No attempt is made to guarantee confinement. The kernel is to provide protection for processes, cevices, whole file systems, and segments, so the size of the protected objects is large ("the grain of protection is coarse").

In addition to the UNIX processes, there will be a scheduler process and an initiator, the former providing data to the kernel assigning execution priorities to the various processes, and the latter replacing the UNIX logon process with code that establishes the protection environment for the user that signs on.

The main problems anticipated are:

(1) The coordination of use of the shared segments between UNIXs. One shared segment will be used for the storage of inodes, file table entries, etc. for each file system. Semaphores may be required in addition to the kernel message facility to coordinate access to these segments.

(2) Constructing an ARPANET NCP or TCP. It is likely that some of this will have to be built into the kernel.

INGRES Data Base Management System

Eugene Wong (Berkeley)

Eugene Wong described the Interactive Graphics and Retrieval System (INGRES) that runs on a PDP 11/40 under UNIX. The current system uses four large processes that call each other sequentially. Some benefit could be obtained from better interprocess communication so that some of the processes could proceed in parallel. (See the ..CC '75 for details on INGRES.)

HRSTS—Harvard-Radcliffe Student Timesharing System

Chuck Prenner (formerly of Harvard, now at Berkeley)

An impressive amount of software has been developed for HRSTS. The following items were mentioned:

1. BASIC: two versions, one based on DEC RT11 BASIC with matrix extensions, another based on DEC RSTS BASIC (which users must first purchase from DEC).

2. MACRO
LINKER—(Version 6 compatible)
DDT—(Brought to RAND by Bob Greenberg)

3. TECO editor

4. FILCON—(like diff)

5. HARVARD SHELLS:
A simplified shell for student use (hand-holding)
Command completion as in TENEX

6. Terminal driver with several interesting features
(a) Page mode for graphics terminals
(b) Setable break characters
(c) The ability to suspend and restart output

7. PPL and ECL i Extensible languages
Manuals can be obtained from:
Center for Research in Computing Techniques

Chuck also spoke of a very good (fast) Fortran that runs under UNIX. It is a modification of DEC Fortran and it is available (after paying DEC a $2500 fee).

The Berkeley 11/70 System

Ken Thompson

Berkeley runs a severely core-limited system, having only 64K words of memory to support an average load of 15 users (22–24 users peak). They use four "home-brew" multiplexers, each having 8 lines. Three of the lines are connected to 11/10 machines used for teaching assembly language programming and interrupt I/O. They have 2 RK05 disks, an RJS04 swapping disk, and 2 RP03 compatible DIVA disks (50 megabytes each).

As usual, Ken had a number of interesting things to say:

1) It appears that the group concept is about to disappear from UNIX, in favor of 16-bit user ids. At Berkeley, the 16-bit ids are partitioned as follows (with a major intent to segregate students from each other, and from the rest of the world)

UID=0:	Super-user (no change)
UID<0:	Student. No reference to other student's files, only "own" files and "public" files, those with owner UID between 1 and 255 (to which normal protections apply).
UID>0:	All others. No reference to students' files. Normal access protection applies to all files.

For class use, the upper 8 bits of the UID is the class number. The teacher has a UID with the low order 8 bits 0, and can access students' files.

2. Disk file quotas: If a directory contains a file named .Q (a quota), then, when the directory inode is in use, the .Q inode is associated with it in the incore inode table. The .Q inode contains the maximum number of blocks that may be used by files in the directory and its decendents, and a count of the number of used blocks. A subdirectory inherits the quota of its ancesters. A new system call was added to make directories with quotas. There were some problems associated with the link operation. It is allowed to exceed quotas temporarily so that the MV operation (rename) cannot cause quotas to be exceeded.

3 A limit was placed on the number of active processes that a user can own. (Enforced in the fork operation: while searching the processor table a count is made.)

4. Ken noted a circumstance under which locks on inodes were not honored within the kernel. He went through the kernel and located a number of other places where the same potential for failure exists. The fixes will appear in the next release.

5. An interesting technique for measuring disk activity was devised. It involves keeping a software busy bit for each of n devices (major or minor) in the low order bits of a word. The word is used as an index into a "time" table of 2**n 32 bit integers. Each time a clock ("uncorrelated" with the scheduling clock) interrupts, the selected "time" entry is incremented. As a result, the "time" entries record the busy time for each combination of devices. By also counting the number of words transferred and the number of transactions on each device, the seek times (including latency and transfer times) can be obtained.

6. Using the technique in 5, Ken determined that for the Berkeley system it was more efficient to use the RJS04 "swapping" disk for the "root" file system (/usr, /tmp, ...) and an RK05 for swapping.

7. A bottleneck was eliminated by allocating two swap buffer headers to avoid putting a process to sleep only to wake it up again to be swapped out.

8. Bell Labs is workng on a C-oriented microprocessor. Get in touch with Sandy Fraser at Bell for information.

9. We learned several things that may improve our response.

    a. Allocate more buffers. Berkeley uses 30, and we are certainly underallocated. A measurement technique for determining buffer utilization was suggested.

    b. The terminal output high and low water marks were set for the 300 baud terminals in use at Bell. With our 9600 baud terminals we should greatly increase these parameters, and the number of character queue elements.

Site Activity Summaries

Steve Holmgren, Center for Adv Comp.—Illinois

Runs an 11/45 with 128k, RP04, RK's, magtape.

Does text processing.

Developed NCP (currently has no server).

Also doing work on local networking.

Mike O'Malley—Berkeley

11/40 system

Runs the Illinois NCP.

Does real time speech processing.

Has added contiguous files.

John Bass—Cal Poly San Luis Obispo

Runs UNIX on an 11/35 (the OEM version of the 40)

Does graphics, data entry, 3-D display for architecture, and language processing.

Dave Farber—UCLA

Has a PASCAL to C translator.

has made changes to ed (e.g., warnings, writeout on hangup).

Jeff Schribman —UCB

Currently has RP03 type disks, DEC RJS04 and RK05's. The RK05 will leave shortly.

Machine connected to CDC 6400.

Runs a RJE station

Harvey Weinstein—Survey Research Center, UCB

Working on computer assisted telephone interviews.

Bill Bridge—Data Disk

Runs an 11/35 with LSI-11 hardware to control terminals.

Developing an automated newspaper production system to run on 11/70's with LSI-11's.

Oliver Roubine—SRI

Has developed a better C debugger.

Art Wilson—UCSD

Has a Fortran which uses 11/40 floating point from Illinois.
Has an 8080 cross assembler
They use the machine for various chemistry applications.

Mike Ubell—UCB

Has GT-40 debugger and Fortran editor.

John Lowry, Carl Sunshine, Steve Zucker—Rand

Runs UNIX on 11/45 with 128K, RP04, RK05's.

Doing intelligent terminal research.

Have developed a 2 dimensional editor, currently being recoded to use ed as a subroutine.

Interested in improved interprocess communication—passed out paper with ideas concerning how this might be accomplished.

Has remind, a Letter version of CROW.

Has file compression routines (25-40% storage savings on typical program and text files).

Runs the Rand NCP (including server).

Mark Kampo—UCLA

Is rewriting tty.o.

Is doing security research.

Has PSCAL, PASCAL, andEUCLID translators.

Has a TENEX compatible HROFF written in C.

Has send message, many new drivers, limited server ftp for the Illinois NCP.

Is connected with coordinated distribution of UNIX software.

TOVAR—Berkeley

Has worked on Dijkstra's primitives (semiphores).

Knows about the CT40.

Has implementd very distant host ARPANET protocol.

Has implemented contiguous files.

Has RK05 comp code for M45.s and dump-interrupting software.

Interested in network graphics program.

Oliver Whitby—SRI

Considering UNIX for scientific communication (Technical Journal Production).

Gary Raetz—Naval Postgraduate School

Working with a variety of display devices.

Has 8080 assembler.

Has a DDT and line editor.

Ira Fuchs—C.U.N.Y.

GT44 system

Runs as text editor front end for 370.

Reported that April 1 of the East Coast Meeting will emphasize graphics.

Discussion Topics

Interprocess Communication (IPC):

Steve Zucker and Carl Sunshine presented the shortcomings of current UNIX IPC facilities. Steve outlined his Port and mpipe implementation. (The two Rand papers had been passed out the day before, but few attendees had had a chance to read them.) There was agreement from several other sites that an ability to wait for multiple inputs would be very desireable, and that mpipes seemed a good way to do this. Bob Fabry was unhappy with the different interface mpipes provided to the reader (the presence of headers).

Other suggstions to facilitate waiting for multiple inputs concentrated on the synchronization problem. Mark Kampo advocated a sleep/wakeup facility for user processes. Steve mentioned the problems of access control, assigning unique wakeup numbers, spurrious wakeups, and implementation with this. Others suggested a more specific facility to "wait" on a specified set of file descriptors and return when the first of them had data available to read. There is also the problem of testing for input (e.g., with EMPTY) and then doing the WAIT, with input arriving between the test and wait. This requires the standard "hyperawake" state kind of resolution when a wakeup to a running process leaves it "hyperawake" so a wait does not block it.

The major implementation problem in current UNIX for both of these suggestions is the difficulty of associating an inode with those processes (readers) "waiting" when a writer writes the inode.

The idea of asychronous, independent I/O channel type operation came up, but this was fairly generally denounced as a tremendous change and undesireable anyway.

Ken Thompson agreed that the current signal facility had a number of problems (but noted that the interrupts did not abort pending disk I/O since the process would be sleeping at a negative priority). Signals had been designed primarily for error or exception condition handling, and not for general interprocess synchronization which Ken was not convinced was a very significant need. By the end of the session he seemed more disposed toward accepting the improved IPC as a legitimate concern, and even willing to consider system changes or additions to provide it.

ARPANET:

Steve Holmgren outlined the ARPANET software design at the University of Illinois. The connection establishment (IPC) code for the NCP has been largely placed in a user process, requiring only 3–5K words (plus buffers) for the remainder of the NCP in the kernel. There is a separate minor device for each network host, and the open command to these network devices defaults to give a Telnet connection. By specifying different options, processes can also get particular sockets, LISTEN mode, or other special actions from the NCP. The server Telnet is still being designed, but will probably feed incoming characters through the TTY drivers.

Multiple-Machine UNIX:

Ken Thompson outlined the dual machine UNIX system he developed at Bell Labs last year using a DR11-C. The basic communication level provides 256 logical channels between two machines. Each data byte is prefixed with a channel number on transmission, and acknowledged by the channel number being received. For more efficient and reliable communication over low bandwidth or long delay lines, the characters on a channel may be buffered into a message with the channel number appearing only once in the header, and a checksum for error detection.

Connections are established between machines by sending control data over channel zero which essentially associates files with channel numbers in each machine. A daemon process which always has a read pending handles the initial connections.

At the user level, commands are interpreted by a Master Shell that looks for a special character (!) in the normal UNIX command syntax which indicates an operation to be performed on the remote machine. Thus !cat rfile > lfile would copy remote file rfile to local file lfile. The Master Shell creates a slave shell process in the remote machine, and all other local and remote processes needed, and sets up all the connections between the local and remote processes.

UNIX Development and Standards:

Because of the consent decree and other intricacies of the UNIX license, Bell is not able or willing to “support” UNIX in the traditional sense of standard releases, updates, fixes to discovered bugs, documentation, etc. There has been talk of ARPA support for a UNIX maintenance and standards project, but this is still uncertain.

There is a strong feeling that sites are going to continue doing their own development in different and sometimes even conflicting directions. There is also a strong frustration with efforts to use each others improvements which often seem to depend on other aspects of the system which are incompatible. A “standard” version of UNIX for reference purposes could make it easier to describe where changes had been made when sites trade software. The clearing house at Chicago Circle mentioned in the February UNIX News may provide this service by sticking a version number on some recent UNIX system and distributing it. But sites currently have widely differing versions of the system (version 5 to version 6 .1) so picking the initial standard is problematic.

Conditional compilation exists (undocumented) in version 6 (plus?) of C, which may help the situation if people “if out” changed standard code rather than deleting it, and “if in” their changes.

Ken Thompson described the way a new version of UNIX gets changed at Bell. The system is constantly undergoing changes, and when these become numerous enough, it is agreed that the manual must be rewritten to reflect the new facilities. In the process of rewriting the manual, everyone remembers all the other things they wanted to do to a particular area of code, and an escalating “frenzy” of activity ensues for a couple of weeks. Finally the manual gets closed and filed one day, after which activity tapers off (but doesn’t cease) because people realize the new changes won’t be known until the next manual rewrite.

Note to our readers
Do you have any back issues of *UNIX News*? Our archive is not complete, and some of the reproductions we have are difficult to read. We’d love to have copies of anything that you have. Contact production@usenix.org.

usenix 40TH ANNIVERSARY 1975–2015

COLUMNS

A Tale of Two Concurrencies (Part 1)

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 (www.swig.org) and Python Lex-Yacc (www.dabeaz.com/ply.html). Beazley is based in Chicago, where he also teaches a variety of Python courses. dave@dabeaz.com

Talk to any Python programmer long enough and eventually the topic of concurrent programming will arise—usually followed by some groans, some incoherent mumbling about the dreaded global interpreter lock (GIL), and a request to change the topic. Yet Python continues to be used in a lot of applications that require concurrent operation whether it is a small Web service or full-fledged application. To support concurrency, Python provides both support for threads and coroutines. However, there is often a lot of confusion surrounding both topics. So in the next two installments, we're going to peel back the covers and take a look at the differences and similarities in the two approaches, with an emphasis on their low-level interaction with the system. The goal is simply to better understand how things work in order to make informed decisions about larger libraries and frameworks.

To get the most out of this article, I suggest that you try the examples yourself. I've tried to strip them down to their bare essentials so there's not so much code—the main purpose is to try some simple experiments. The article assumes the use of Python 3.3 or newer.

First, Some Socket Programming

To start our exploration, let's begin with a little bit of network programming. Here's an example of a simple TCP server implemented using Python's low-level `socket` module [1]:

```
# server.py
from socket import *
def tcp_server(address, handler):
    sock = socket(AF_INET, SOCK_STREAM)
    sock.setsockopt(SOL_SOCKET, SO_REUSEADDR, 1)
    sock.bind(address)
    sock.listen(5)
    while True:
        client, addr = sock.accept()
        handler(client, addr)
def echo_handler(client, address):
    print('Connection from', address)
    while True:
        data = client.recv(1000)
        if not data:
            break
        client.send(data)
    print('Connection closed')
    client.close()
if __name__ == '__main__':
    tcp_server(('',25000), echo_handler)
```

Run this program in its own terminal window. Next, try connecting to it using a command such as `nc 127.0.0.1 25000` or `telnet 127.0.0.1 25000`. You should see the server echoing what you type back to you. However, open up another terminal window and try repeating the `nc` or `telnet` command. Now you'll see nothing happening. This is because the server only supports a single client. No support has been added to make it manage multiple simultaneous connections.

Programming with Threads

One way to support concurrency is to use the built-in `threading` library [2]. Simply change the `tcp_server()` function to launch a new thread for each new connection as follows:

```
# server.py
from socket import *
from threading import Thread
def tcp_server(address, handler):
    sock = socket(AF_INET, SOCK_STREAM)
    sock.setsockopt(SOL_SOCKET, SO_REUSEADDR, 1)
    sock.bind(address)
    sock.listen(5)
    while True:
        client, addr = sock.accept()
        t = Thread(target=handler, args=(client, addr))
        t.daemon=True
        t.start()
...
```

That's it. If you repeat the connection experiment, you'll find that multiple clients work perfectly fine.

Under the covers, a `Thread` represents a concurrently executing Python callable. On UNIX, threads are implemented using POSIX threads and are fully managed by the operating system. A common confusion concerning Python is a belief that it uses some kind of "fake" threading model—not true. Threads in Python are the same first-class citizens as you would find in C, Java, or any number of programming languages with threads. In fact, Python has supported threads for most of its existence (the first implementations provided thread support on Irix and Solaris around 1992). There is also support for various synchronization primitives such as locks, semaphores, events, condition variables, and more. However, the focus of this article is not on concurrent programming algorithms per se, so we're not going to focus further on that.

The Global Interpreter Lock

The biggest downside to using threads in Python is that although the interpreter allows for concurrent execution, it does not support parallel execution of multiple threads on multiple CPU cores. Internally, there is a global interpreter lock (GIL) that synchronizes threads and limits their execution to a single core (see [3] for a detailed description of how it works). There are various reasons for the existence of the GIL, but most of them relate to aspects of the implementation of the Python interpreter itself. For example, the use of reference-counting-based garbage collection is poorly suited for multithreaded execution (since all reference count updates must be locked). Also, Python has historically been used to call out to C extensions that may or may not be thread-safe themselves. So the GIL provides an element of safety.

For tasks that are mostly based on I/O processing, the restriction imposed by the GIL is rarely critical—such programs spend most of their time waiting for I/O events, and waiting works just as well on one CPU as on many. The major concern is in programs that need to perform a significant amount of CPU processing. To see this in action, let's make a slightly modified server that, instead of echoing data, computes Fibonacci numbers using a particularly inefficient algorithm:

```
# server.py
...
def fib(n):
    if n <= 2:
      return 1
    else:
      return fib(n-1) + fib(n-2)
def fib_handler(client, addr):
    print('Connection from', address)
    while True:
        data = client.recv(1000)
        if not data:
            break
        result = fib(int(data))
        client.send(str(result).encode('ascii')+b'\n')
    print('Connection closed')
    client.close()
if __name__ == '__main__':
    tcp_server(('',25000), fib_handler)
```

If you try this example and connect using `nc` or `telnet`, you should be able to type a number as input and get a Fibonacci number returned as a result. For example:

```
bash % nc 127.0.0.1 25000
10
55
20
6765
```

Larger numbers take longer and longer to compute. For example, if you enter a value such as 40, it might take as long as a minute to finish.

Now, let's write a little performance test. The purpose of this test is to repeatedly hit the server with a CPU-intensive request and to measure its response time.

```
# perf1.py
from socket import *
import time
sock = socket(AF_INET, SOCK_STREAM)
sock.connect(('127.0.0.1', 25000))
while True:
    start = time.time()
    sock.send(b'30')
    resp = sock.recv(100)
    end = time.time()
    print(end-start)
```

Try running this program. It should start producing a series of timing measurements such as this:

```
bash % python3 perf1.py
0.6157200336456299
0.6006970405578613
0.6721141338348389
0.7784650325775146
0.5988950729370117
...
```

Go to a different terminal window and run the same performance test as the first one runs (you have two clients making requests to the server). Carefully watch the reported times. You'll see them suddenly double like this:

```
0.6514902114868164
0.629213809967041
1.2837769985198975          # 2nd test started
1.4181411266326904
1.3628699779510498
...
```

This happens even if you're on a machine with multiple CPU cores. The reason? The global interpreter lock. Python is limited to one CPU, so both clients are forced to share cycles.

Lesser Known Evils of the GIL

The restriction of execution to a single CPU core is the most widely known evil of the GIL. However, there are two other lesser known facets to it that are worth considering. The first concerns the instructions that Python executes under the covers. Consider a simple Python function:

```
def countdown(n):
    while n > 0:
        print('T-minus', n)
        n -= 1
```

To execute the function, it is compiled down to a Python-specific machine code that you can view using the `dis.dis()` function:

```
>>> import dis
>>> dis.dis(countdown)
  2         0 SETUP_LOOP          39 (to 42)
      >>    3 LOAD_FAST            0 (n)
            6 LOAD_CONST           1 (0)
            9 COMPARE_OP           4 (>)
           12 POP_JUMP_IF_FALSE   41
  3        15 LOAD_GLOBAL          0 (print)
           18 LOAD_CONST           2 ('T-minus')
           21 LOAD_FAST            0 (n)
           24 CALL_FUNCTION        2 (2 positional, 0 keyword pair)
           27 POP_TOP
  4        28 LOAD_FAST            0 (n)
           31 LOAD_CONST           3 (1)
           34 INPLACE_SUBTRACT
           35 STORE_FAST           0 (n)
           38 JUMP_ABSOLUTE        3
      >>   41 POP_BLOCK
      >>   42 LOAD_CONST           0 (None)
           45  RETURN_VALUE
```

In this code, each low-level instruction executes atomically. That is, each instruction is uninterruptible and can't be preempted. Although most instructions execute very quickly, there are edge cases where a single instruction might take a very long time to execute. To see that, try this experiment where you first launch a simple thread that simply prints a message every few seconds:

```
>>> import time
>>> def hello():
...     while True:
...         print('Hello')
...         time.sleep(5)
...
>>> import threading
>>> t = threading.Thread(target=hello)
>>> t.daemon=True
>>> t.start()
Hello
Hello
... repeats every 5 seconds
```

Now, while that thread is running, type the following commands into the interactive interpreter (note: it might be a bit weird since "Hello" is being printed at the same time).

```
>>> nums = range(1000000000) # Use xrange on Python 2
>>> 'spam' in nums
```

At this point, the Python interpreter will go completely silent. You'll see no output from the thread. You'll also find that not even the Control-C works to interrupt the program. The reason is that the “in” operator in this example is executing as a single interpreter instruction—it just happens to be taking a very long time to execute due to the size of the data. In practice, it’s actually quite difficult to stall pure-Python code in this way. Ironically, it’s most likely to occur in code that calls out to long-running operations implemented in C extension modules. Unless the author of an extension module has programmed it to explicitly release the GIL, long operations will stall everything else until they complete (see [4] for information on avoiding this).

The second subtle problem with the GIL is that it makes Python prioritize long-running CPU-bound tasks over short-running I/O-bound tasks. To see this, type in the following test program, which measures how many requests are made per second:

```
# perf2.py
import threading
import time
from socket import *
sock = socket(AF_INET, SOCK_STREAM)
sock.connect(('127.0.0.1', 25000))
N = 0
def monitor():
    global N
    while True:
        time.sleep(1)
        print(N, 'requests/second')
        N = 0
t = threading.Thread(target=monitor)
t.daemon=True
t.start()
while True:
     sock.send(b'1')
     resp = sock.recv(100)
     N += 1
```

This program hits the server with a rapid-fire stream of requests. Start your server and run it; you should see output such as this:

```
bash % python3 perf2.py
22114 requests/second
21874 requests/second
21734 requests/second
21137 requests/second
21866 requests/second
...
```

While that is running, initiate a separate session and try computing a large Fibonacci number:

```
bash % nc 127.0.0.1 25000
40
102334155     (takes awhile to appear)
```

When you do this, the `perf2.py` program will have its request rate drop precipitously like this:

```
21451 requests/second
21913 requests/second
6942 requests/second
99 requests/second
103 requests/second
101 requests/second
99 requests/second
...
```

This more than 99% drop in the request rate is due to the fact that if any thread wants to execute, it waits as long as 5 ms before trying to preempt the currently executing thread. However, if any computationally intensive task is running, it will almost always be holding the CPU for the entire 5 ms period and stall progress on short I/O intensive tasks that want to run.

Both of these problems, uninterruptible instructions and prioritization of CPU-bound work, would manifest themselves in an application as a kind of performance “glitch” or a kind of sluggishness. For example, suppose that this service was implementing a backend service for a Web site. Maybe most of the operations are fast-running data queries, but suppose that there were a few corner cases where the service had to perform a significant amount of CPU processing. For those cases, you would find that the responsiveness of the service would degrade significantly as those CPU-intensive tasks are carried out.

Personally, I think the inversion of priority of CPU-bound threads over I/O-bound threads might be the most serious problem with using Python threads—more so than the limitation of execution to a single CPU core. This preference for CPU-bound tasks is exactly the opposite of how operating systems typically prioritize processes (short-running interactive processes usually get priority). In most applications, it’s almost always better to maintain a quick response time even if it means certain long-running operations take a slight bit longer to complete than they already do.

Using Subprocesses

Although you might look in dismay at the performance of Python threads, keep in mind that their primary limitation concerns long-running CPU-bound tasks. If this applies, you’ll want to seek some other subdivision of tasks in your system. A typical solution is to run multiple instances of the Python interpreter, either through process forking or the use of a process pool.

For example, to use a process pool, you can modify the server to use the `concurrent.futures` module as follows:

```
# server.py
from concurrent.futures import ProcessPoolExecutor as Pool
NPROCS = 4
pool = Pool(NPROCS)
def fib_handler(client, address):
    print('Connection from', address)
    while True:
        data = client.recv(1000)
        if not data:
            break
        future = pool.submit(fib, int(data))
        result = future.result()
        client.send(str(result).encode('ascii')+b'\n')
    print('Connection closed')
    client.close()
...
```

If you make this simple change, you'll find that the first performance test (perf1.py) now nicely scales to use all available CPU cores.

However, using a process Pool for short-running tasks is probably not the best approach. If you run the second performance test (perf2.py), you'll see about a 95% reduction in the request rate such as this:

```
bash % python3 perf2.py
1319 reqs/sec
1313 reqs/sec
1315 reqs/sec
1308 reqs/sec
```

This performance reduction is solely due to all of the extra overhead associated with sending the request to another process, serializing data, and so forth. If there's any bright spot, it's that this request rate will now remain constant even if another client starts performing a long-running task (so, at the very least, the performance will simply be consistently and predictably bad). A smarter approach might involve a threshold that only kicks work out to a pool if it's known in advance that it will take a long time to compute. For example:

```
# server.py
...
def fib_handler(client, address):
    print('Connection from', address)
    while True:
        data = client.recv(1000)
        if not data:
            break
        n = int(data)
        if n > 15:
            future = pool.submit(fib, n)
            result = future.result()
        else:
            result = fib(n)
        client.send(str(result).encode('ascii')+b'\n')
    print('Connection closed')
    client.close()
...
```

Using a process pool is not the only approach. For example, an alternative approach might involve a pre-forked server like this:

```
# server.py
from socket import *
import os
NPROCS = 8
def tcp_server(address, handler):
    sock = socket(AF_INET, SOCK_STREAM)
    sock.setsockopt(SOL_SOCKET, SO_REUSEADDR, 1)
    sock.bind(address)
    sock.listen(5)
    # Fork copies of the server
    for n in range(NPROCS):
        if os.fork() == 0:
            break
    while True:
        client, addr = sock.accept()
        handler(client, addr)
...
```

In this case, there is no communication overhead associated with submitting work out to a pool. However, you're also strictly limiting your concurrency to the number of processes spawned. So if a large number of long-running requests were made, they might lock everything out of the server until they finish.

Memory Overhead of Threads

A common complaint lodged against threads is that they impose a steep memory overhead. To be sure, if you create 1000 threads, each thread requires some dedicated memory to use as a call stack and some data structures for management in the operating system. However, this overhead is intrinsic to POSIX threads and not to Python specifically. The Python-specific overhead is actually quite small. Internally, each Python thread is represented by a small C data structure that is less than 200 bytes in size. Beyond that, the only additional overhead are the stack frames used to make function calls in the thread.

Even the apparent memory overhead of threads can be deceptive. For example, by default, each thread might be given 8 MB of memory for its stack (thus, in a program with 1000 threads, it will appear that more than 8 GB of RAM is being used). However, it's important to realize that this memory allocation is typically just a maximum reservation of virtual memory, not actual physical RAM. The operating system will allocate page-sized chunks of memory (typically 4096 bytes) to this space as it's needed during execution but leave the unused space unmapped. Many operating systems (e.g., Linux) take it a step further and won't even reserve space on the swap disk for thread stacks unless specifically configured [5]. So the actual memory overhead of threads is far less than it might seem at first glance. (Note: the lack of a swap allocation for threads presents a possible danger to production systems—if memory is ever exhausted, the system might start randomly killing processes to make space!)

As far as Python is concerned, the main memory overhead risk is in code based on deeply recursive algorithms because this would create the potential to use up all of the thread stack space. However, most Python programmers just don't write code like this. In fact, if you blow up the stack on purpose, you'll find that it takes nearly 15,000 levels of recursive function calls to do it. The bottom line: it's unlikely that you would need to worry about this in normal code. In addition, if you're really concerned, you can set a much smaller thread stack size using the `threading.stack_size(nbytes)` function.

All of this said, the overhead of threads is still a real concern. Many systems place an upper limit on the number of threads a single process or user can create. There are also certain kinds of applications where the degree of concurrency is so high that threads simply aren't practical. For example, if you're writing a server to support something like WebSockets, you might have a scenario where the system needs to maintain tens of thousands of open socket connections simultaneously. In that case, you probably don't want to manage each socket in its own thread (we'll address this in the next installment).

Everything Is Terrible, Well, Only Sometimes. Maybe.

If you've made it this far, you might be inclined to think that just about everything with Python threads is terrible. To be sure, threads are probably not the best way to handle CPU-intensive work in Python, and they might not be appropriate if your problem involves managing 30,000 open socket connections. However, for everything else in the middle, they offer a sensible choice.

For one, threads work great with programs that are primarily performing I/O. For example, if you're simply moving data around on network connections, manipulating files, or interacting with a database, most of your program's time is going to be spent waiting around. So you're unlikely to see the worst effects of the GIL. Second, threads offer a relatively simple programming model. Launching a thread is easy: they are generally compatible with most Python code that you're likely to write, and they're likely to work with most Python extensions (caveat: if you're manipulating shared state, be prepared to add locks). Finally, Python provides libraries for moving work off to separate processes if you need to.

In the next installment, we'll look at an alternative to programming with threads based on Python coroutines. Coroutines are the basis of Python's new `asyncio` module, and the techniques are being used a variety of other programming languages as well.

References

[1] Python3 socket module: https://docs.python.org/3/library/socket.html.

[2] Python3 threading module: https://docs.python.org/3/library/threading.html.

[3] Dave Beazley, "Understanding the Python GIL": http:/www.dabeaz.com/GIL.

[4] Thread state and the Global Interpreter Lock: https://docs.python.org/3/c-api/init.html#thread-state-and-the-global-interpreter-lock.

[5] R. Love, "Linux System Programming," 2nd ed. (O'Reilly Media, Inc., 2013); see the section on "Opportunistic Allocation" in Chapter 9.

Practical Perl Tools

Parallel Asynchronicity, Part 1

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 thirty 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

At some point everyone gets the desire to be able to do multiple things at once or be in multiple places at the same time as a way of getting more done. And although we try to multitask, the research keeps piling up to suggest that humans aren't so good at intentional multitasking. But computers, they do a much better job at this, that is, if we humans can express clearly just how we want them to work on multiple things at once. I thought it might be fun to explore the various ways we can use Perl to write code that performs multiple tasks at once. This can be a fairly wide-ranging topic, so we're going to take it on over multiple columns to give us plenty of space to peruse the subject. Note: for those of you with photographic memories, I touched on a similar subject in this column back in 2007. There will be some overlap (and I might even quote myself), but I'll be bringing the topic up to date by bringing in modules that weren't around in the good ole days.

One quick caveat that my conscience forces me to mention: to avoid writing a book on the topic (been there, done that), these columns will use UNIX or UNIX-derivative operating systems as their reference platform. There's been lots of great work done over the years for other platforms (I'm looking at you Windows), but I won't be making any guarantees that everything written here works on anything but a UNIX-esque system. Caveat Microsoft Emptor and all of that.

Fork!

Usually I don't like to get to forking without a little bit of warm up, but that is indeed the simplest way to get into the parallel processing game. Perl has a fork() function that lets us spin off another copy of a running Perl interpreter that continues to execute the current program. Let's see how it works.

If the Perl interpreter encounters the first line of the program below:

```
my $pid = fork();
print $pid,"\n";
```

a second Perl interpreter comes into being (i.e., a copy is forked off the running interpreter) that is also running the program. That second copy is referred to as a child of the original copy (which, as you would expect, is called the parent process). From the perspective of the program itself, neither copy can tell that anything special has occurred (they are both running the exact same program, have the same file descriptors open, etc.) with one very small difference: when the fork() program line has successfully finished executing, $pid in the *parent* process gets set to the process ID of the child process. In the child process, $pid will be set to 0. So if I run this program as is, I'll get output like this:

```
$ perl fork.pl
6240
0
```

The parent has printed the process ID of the child process (6240), the child printed 0 as expected. As an important aside for your programming, if for some reason the fork() call fails, $pid will be undef in the parent (and yes, you should test for that happening).

The reason this matters is that a parent process has a responsibility for a task in addition to any other work it plans to do that a child does not. A parent process is responsible for "reaping" its children after they exit lest they remain zombies (and we all know zombies are entertaining, I mean bad, right?). Wikipedia has a great description of this [1]:

> On Unix and Unix-like computer operating systems, a zombie process or defunct process is a process that has completed execution (via the exit system call) but still has an entry in the process table: it is a process in the "Terminated state". This occurs for child processes, where the entry is still needed to allow the parent process to read its child's exit status: once the exit status is read via the wait system call, the zombie's entry is removed from the process table and it is said to be "reaped".

There are two common ways for reaping: manually or signal-based. The manual way is the most straightforward. The parent will call wait() or waitpid(), which will block until the child exits. The code looks something like this:

```
my $pid = fork();

die "fork failed!: $!\n";  if (!defined $pid);

if ($pid == 0) {
   # I'm a client
   # do stuff
   exit 0;
}
else {
   # I'm the parent
   # can also do stuff, then reap the child
   waitpid $pid, 0;
}
```

Now, that code just shows a single fork. If we wanted a parent to fork repeatedly, that is as easy as putting a while loop around the fork() call and either keep track of the child process IDs (so we can have the parent wait for each explicitly with waitpid()) or have a similar loop at the end of the parent script that repeatedly calls wait() (which just waits for any child process) the right number of times to clean up after each fork().

If you don't like this approach, another one is to make use of the fact that the parent process should receive a signal when each child process exits (SIGCHLD to be precise). If we add a signal handler that either ignores the signal explicitly or reaps the child that signaled it, we avoid zombies as well. If you plan to go this route, I recommend looking up the Perl Cookbook recipe on "Avoiding Zombie Processes" because it does a good job of laying out some of the caveats you'll need to know.

One last tip for you if you plan to write manually forking code: I've seen far too many fork bombs (where a process forks itself and the machine it is on into oblivion) in my time. Please put logic into your code that limits and/or prevents unbridled forking. Keep a counter, check for a sentinel file (i.e., if a file with a name on disk exists, don't fork), create a dead man's switch (only fork if a file or some other condition is present), and so on. Any strategy to avoid this problem is likely better than no strategy.

Let Someone Else Fork for You

Truth be told, I haven't written code that calls fork() in quite a few years. Ever since I discovered a particular module and how easy it let me spread "run this in parallel" pixie dust on my previously serial code, I really haven't bothered with any of that fork() / wait() drudgery. The module I speak of and love dearly is Parallel::ForkManager. Looking back at the 2007 column, it is clear that my affections haven't waned over that time because the example I gave then is still pertinent today.

Let's quickly revisit that code. At the time, I mentioned having a very large directory tree that I needed to copy from one file system to another. I needed to copy each subdirectory over separately using code similar to this:

```
opendir( DIR, $startdir ) or
   die "unable to open $startdir:$!\n";
while ( $_ = readdir(DIR) ) {
    next if $_ eq ".";
    next if $_ eq "..";
    push( @dirs, $_ );
}
closedir(DIR);

foreach my $dir ( sort @dirs ) {
    ( do the rsync );
}
```

Since the copy operations are not related to each other (except by virtue of touching the same file servers), we could run the copies in parallel. But we have to be a little careful—we probably don't want to perform the task in the maximally parallel fashion (i.e., start up *N* rsyncs where *N* is the number of subdirectories) because that is sure to cause too much I/O and perhaps memory and CPU contention. We'll want to run with a limited number of copies going at a time. Here's some revised code that uses Parallel::ForkManager:

```
# ...read in the list of subdirectories as before

my $pm = new Parallel::ForkManager(5);

foreach my $dir (sort @dirs){
    # we are a child process if we get past this line
    $pm->start and next;

    ( ... do the rsync ... );

    $pm->finish; # terminate child process
}

  # hang out until all processes have completed
  $pm->wait_all_children;
```

Let's take a walk through the code. The first thing we do is instantiate a Parallel::ForkManager object. When we do, we provide the maximum number of processes we want running at a time (five in this case). We then iterate over each subdirectory we are going to copy. For each directory, we use start() to fork a new process. If the max number of processes is already running, this command will pause until there is a free spot. If we are able to fork(), the line $pm->start returns the pid of the child process to the parent process and 0 to the child as in our last section.

The logic of the "and next" part of the line is a little tricky so let me go to super slow-mo and explain what is going on very carefully. We need to understand two cases: what happens to the parent and what happens to the child process.

If we are the parent process, we'll get a process ID of the child back from start(), and so the line will become something like

```
6240 and next;
```

Because 6240 is considered a "true" value in Perl, the next statement will run and the rest of the contents of the loop will be skipped. This lets the parent move on to the "next" subdirectory so it can start() the next one in the list.

If we are the child process, we'll get a 0 back from the start() call, so the line becomes

```
0 and next;
```

Since Perl short-circuits the "and" construct when it knows the first value is false (as it is here), next isn't called so the contents of the loop (the actual rsync) is run by the child process. The child process then calls $pm->finish to indicate it is done and ready to exit.

At the very end, we call wait_all_children in the parent process so it will hang out to reap the children that were spawned in the process.

As I mentioned in 2007, all it takes is four additional lines for the program to run my actions in parallel, keeping just the right number of processes going at the same time. Easy peasey.

Better Threads

After process forking, the very next topic that usually comes up in a discussion of parallel processing is threads. The usual idea behind threads is they are lightweight entities that live within a single process. They are considered lightweight because they don't require a new process (e.g., with all of the requirements of running a new Perl interpreter) to be spun up for each worker. As a quick aside: modern operating systems do a bunch of fancy tricks to make process spawning/forking not as resource intensive as might first appear, but it still is likely to be heavier than decent threading support. A threading model can sometimes make the programmer work a bit harder for reasons we'll see in the next installment, but it is often worth it.

Allow me to surprise you by ignoring any of the built-in Perl threading support and instead moving on to a module that I think provides for more pleasant use of threads under Perl: Coro. I'll let an excerpt from the module's intro doc explain [2]:

> Coro started as a simple module that implemented a specific form of first class continuations called Coroutines. These basically allow you to capture the current point execution and jump to another point, while allowing you to return at any time, as kind of non-local jump, not unlike C's "setjmp/longjmp."...
>
> One natural application for these is to include a scheduler, resulting in cooperative threads, which is the main use case for Coro today....
>
> A thread is very much like a stripped-down perl interpreter, or a process: Unlike a full interpreter process, a thread doesn't have its own variable or code namespaces—everything is shared. That means that when one thread modifies a variable (or any value, e.g., through a reference), then other threads immediately see this change when they look at the same variable or location.
>
> Cooperative means that these threads must cooperate with each other, when it comes to CPU usage—only one thread ever has the CPU, and if another thread wants the CPU, the running thread has to give it up. The latter is either explicitly, by calling a function to do so, or implicitly, when waiting on a resource (such as a Semaphore, or the completion of some I/O request).

Coro will allow us to write a program where various parts of the program can do some work and then hand off control of the CPU to other parts. If that sounds a lot like a subroutine to you, you are having the same reaction I did when I first started to learn about Coro. One thing that helped me was this comparison between subroutines and coroutines in Wikipedia [3]:

> When subroutines are invoked, execution begins at the start, and once a subroutine exits, it is finished; an instance of a subroutine only returns once, and does not hold state between invocations. By contrast, coroutines can exit by calling other coroutines, which may later return to the point where they were invoked in the original coroutine; from the coroutine's point of view, it is not exiting but calling another coroutine. Thus, a coroutine instance holds state, and varies between invocations; there can be multiple instances of a given coroutine at once. The difference between calling another coroutine by means of "yielding" to it and simply calling another routine (which then, also, would return to the original point) is that the latter is entered in the same continuous manner as the former. The relation between two coroutines which yield to each other is not that of caller-callee, but instead symmetric.

I can't believe I'm going to use this analogy, but it may help you think of this model like a group of people standing in a circle playing Hacky Sack™. One person starts with the footbag, does some tricks, and then (in order for the game to be interesting to all involved) has to pass the bag to another participant who does whatever tricks he or she knows. That person does a few things and then passes it along to the next person and so on. If you could speed up the game such that all of the tricks and all of the passes happen fast enough, you would get to watch a pretty entertaining blur of activity consisting of multiple tasks appearing to basically happen at the same time. This analogy breaks down when you start to talk about the various ways you can synchronize cooperative threads, but it at least gets you started.

We are starting to come to the end of this column (I know, just when it was starting to get good), but before we go, let's learn the very basics of how to use Coro at sort of the "hacky sack analogy" level. Next time we'll pick up right from these basics and look at the more sophisticated features surrounding synchronization and data passing between threads.

The basic way to define a thread in Coro is to use the async function. This function looks almost exactly like a subroutine definition except arguments are passed after the code block:

```
use Coro;
async { print "hi there $_[0]\n"; } 'usenix';
```

So what do you imagine that code prints? If you said "nothing!" you win. If that answer makes you shake your head in confusion, don't worry, it is a bit of a trick question if you haven't worked with this package before. Let me explain.

When the program starts, it is running in what we'll call the "main" thread. This thread runs the async command you see above to queue the requested code as a new thread. Then the main thread exits because there is nothing left in the program to run. As a result, the thread it queued up never got a chance (sob) to run, hence no output. If we wanted that new thread to get some CPU time, we have to give up the CPU in the main thread, as in:

```
use Coro;
async { print "hi there $_[0]\n"; } 'usenix';
cede;
```

Now we get the "hi there usenix" output we expect. We can yield the CPU (which is what most thread packages call it instead of cede) in any thread we want. Let's play around with this idea a bit. What would you guess this program returns? (Warning—it is another trick question.)

```
use Coro;

async {
    print "1\n";
    cede;
    print "back to 1\n";
};
async {
    print "2\n";
    cede;
    print "back to 2\n";
};
async {
    print "3\n";
    cede;

};
cede;
```

Here's the answer:

```
1
2
3
```

Let's walk through what is going on. The main thread starts. It queues thread 1 to run, then thread 2, then thread 3. Finally, the main thread cedes control of the CPU to the next thing that is ready to run, which happens to be thread 1. Thread 1 prints "1" and then cedes control to the next thing in the queue (thread 2). This repeats until thread 3, which cedes control back to the main thread. The main thread has nothing more to do, so the program exits.

If we wanted to return to any of the threads so they can continue and print their second line, the main thread would have to explicitly cede control back to them again. This is just as easy as adding an extra “cede;” to the end of the program. If you are not used to thinking “what is currently running? what could be running?” it can take a little getting used to. Luckily, there are ways to debug Coro programs. We’ll talk about that and other advanced subjects in the next installment. Take care, and I’ll see you next time.

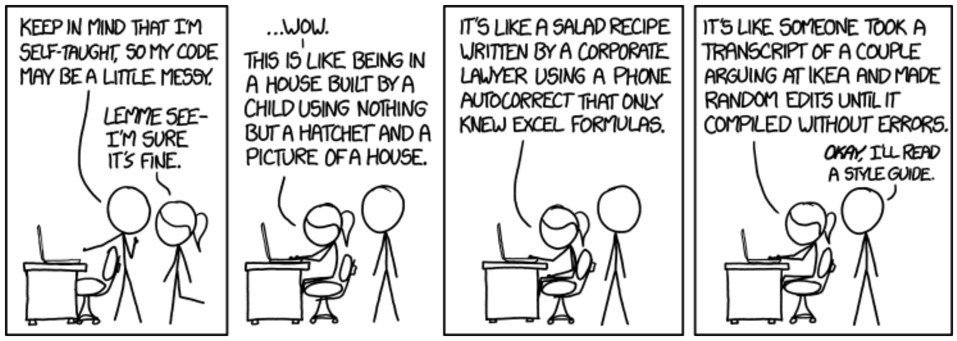
References

[1] Wikipedia, “Zombie process”: https://en.wikipedia.org/wiki/Zombie_process.

[2] Coro: http://search.cpan.org/dist/Coro/coro/intro.pod.

[3] Wikipedia, “Coroutines”: https://en.wikipedia.org/wiki/Coroutine.

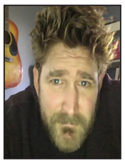
XKCD


KEEP IN MIND THAT I'M SELF-TAUGHT, SO MY CODE MAY BE A LITTLE MESSY.
LEMME SEE- I'M SURE IT'S FINE.
...WOW.
THIS IS LIKE BEING IN A HOUSE BUILT BY A CHILD USING NOTHING BUT A HATCHET AND A PICTURE OF A HOUSE.
IT'S LIKE A SALAD RECIPE WRITTEN BY A CORPORATE LAWYER USING A PHONE AUTOCORRECT THAT ONLY KNEW EXCEL FORMULAS.
IT'S LIKE SOMEONE TOOK A TRANSCRIPT OF A COUPLE ARGUING AT IKEA AND MADE RANDOM EDITS UNTIL IT COMPILED WITHOUT ERRORS.
OKAY, I'LL READ A STYLE GUIDE.

xkcd.com

iVoyeur
Bridge Building

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

For the last year I've worn the title "Developer Evangelist," which aside from the anxiety you'd expect to accompany any large, sudden, and ill-considered career transformation, has been a fantastic experience.

I'd estimate that about a third of my time is spent in a sort of forensic exploration of my own engineering team. I comb through GitHub commit logs, chatroom scrollback, and sometimes interview my coworkers in search of narratives that I think might become good conference talks, or blog posts. Yes. They actually pay me to do this.

Once I have a story I think is interesting, I work up a proposal that describes it, and then I go about trying to find a conference that would be a good fit for the story. A talk about how we arrived at our data storage schema might interest the attendees of LISA, while a story about a microservices re-architecture might be more interesting to the audience at Velocity. I submitted over 40 CFPs in 2014, and wound up giving 12 conference talks last year—a staggering number, at least to me—and every single time I was surprised by what the conference organizers accepted and rejected.

Being an interloper in their midst—someone invited from the outside to tell a story—accentuates the otherness between us and awakens in me the anthropological observer. I see micro-community everywhere these days, a side effect of my almost weekly personal interaction with these regional, fleeting, ad hoc flocks of people who compute, but in some ever slight but critically important way differ from how I compute.

Entire conferences of Software Architects, or people who Ruby. Pythonists focused on data science, or PHP mobile Web developers, or software engineers fighting for diversity, all the same, and yet drastically different. Like Bedouin brought suddenly together by the hundreds or thousands for a few short days. The punctuated equilibrium of nerd community evolution. Each event feels so rare, and yet there are so many nerd microcommunities that their gatherings occur by the thousands each year. Even among each of them, there are smaller micro-communities—groups that cluster around this or that tool, methodology, or model of thought.

When Turing wrote about universal machines, I wonder whether he had any notion of the heterogeneity of human social interactions they would foster. Speaking of universal machines, consider the JVM, for example. Not only does the JVM itself have its own conference [1], and therefore its own community, but nearly 20 different programming languages run on top of the JVM [2], almost all of which have their own conference and community.

There's certainly some overlap, but it's difficult to imagine one person who is both an avid participant in the Rhino AND Jacl communities (much less a JVM uber-linguist who participates in them all). So we can think of the JVM as a sun at the center of a solar system of communities. Many of which, like JRuby, simultaneously orbit other stars.

Here is yet another way that the JVM is a fascinating piece of software: it creates tribalism by encouraging dissent and competition with respect to things like language semantics, while at the same time giving everyone a common ground to stand on by abstracting away the uncontroversial: memory management, parallelism, and garbage collection.

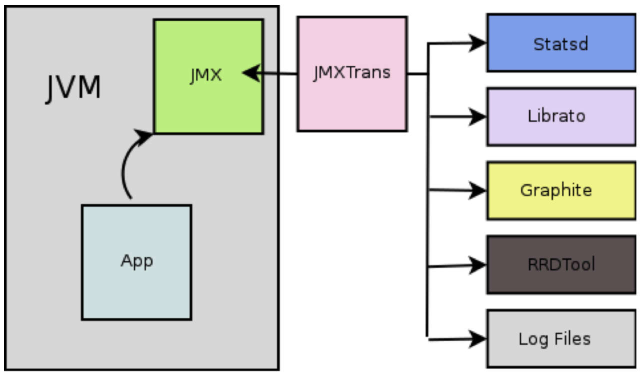


Figure 1: Jmxtrans follows the centralized polling pattern.

Given the (supposed) topic of this column, monitoring might have occurred to you as another common ground the JVM languages share. To be sure, JMX provides a single means of extracting monitoring and metrics data from any language or application that runs on the JVM. But monitoring is also a community of its own, with its own fair share of dissent and competition. This month, I thought we'd look at jmxtrans, a small tool that bridges these worlds, by extracting JVM metrics and shipping them to the monitoring tool of your choice.

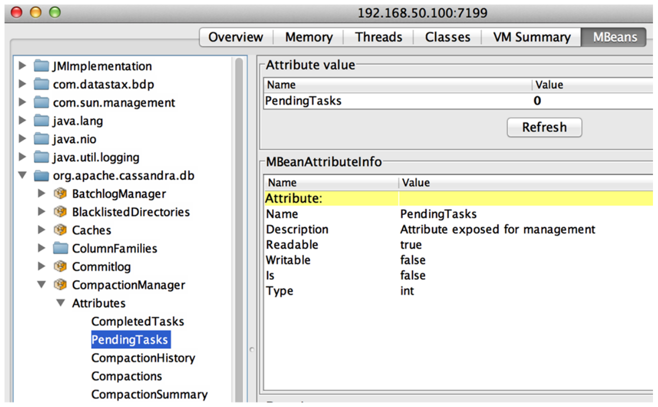
In its own words, jmxtrans is: "the missing connector between speaking to a JVM via JMX on one end and whatever logging/monitoring/graphing package that you can dream up on the other." If you aren't familiar, Java Management Extensions (JMX) is the formally provided channel for exporting metrics from applications running inside the JVM to other processes. Jmxtrans acts as a glue-layer between an application that exposes data via JMX and various monitoring and metrics tools that import and make use of that data.

As you can see in Figure 1, jmxtrans is a specialized centralized poller. It periodically polls a running JMX process, grabbing the metrics you're interested in and emitting them (via a series of output writers) to your monitoring system. Because it polls JMX, it obviously requires that JMX be enabled on the JVM that's running your application.

You can enable JMX by specifying several options to the JVM when you start it. The simplest (and also most insecure) configuration consists of these four options:

- Djava.rmi.server.hostname= <IP Address>
- Dcom.sun.management.jmxremote.rmi.port= <PORT>
- Dcom.sun.management.jmxremote.ssl=false
- Dcom.sun.management.jmxremote.authenticate=false

A minimally useful configuration should use authentication and SSL, both of which depend on your environment. See the official JMX documentation for secure configuration information [3].



```
▶ JMImplementation
▶ com.datastax.bdp
▶ com.sun.management
▶ java.lang
▶ java.nio
▶ java.util.logging
▼ org.apache.cassandra.db
    ▶ BatchlogManager
    ▶ BlacklistedDirectories
    ▶ Caches
    ▶ ColumnFamilies
    ▶ Commitlog
    ▼ CompactionManager
        ▼ Attributes
            CompletedTasks
            PendingTasks
            CompactionHistory
            Compactions
            CompactionSummary
```

Attribute value

Name	Value
PendingTasks	0

MBeanAttributeInfo

Name	Value
Attribute:	
Name	PendingTasks
Description	Attribute exposed for management
Readable	true
Writable	false
Is	false
Type	int

Figure 2: Exploring Cassandra's JMX metrics with JConsole

Once enabled, the JMX service will be available on the IP and port that you specify. Many different tools can interact with a running JMX service, including jmxtrans, jvisualvm, and JConsole.

Because the specific metrics available depend on what your application exports, you'll need to use a tool like JConsole to explore the JMX service. I'm using it in Figure 2 to explore some of the metrics exported by an Apache Cassandra server with JMX enabled. JConsole is included with the JDK, so you shouldn't need to download or install anything to get it running. Connect to a running JMX instance with:

```
jconsole <IP>:<PORT>
```

The names in the main window will correspond to the metric names I'll refer to later when I configure jmxtrans. The hierarchical list in the left windowpane corresponds to jmxtrans's obj configuration attribute, while the list on the right corresponds to attr names.

Even if your application doesn't export any metrics at all, JMX will still emit helpful memory-management, CPU, and thread-usage metrics from the JVM, as you can see in Figure 3. These metrics are otherwise difficult to obtain from a running JVM.

Once you have JMX enabled for the application you want to monitor, and you've decided on the metrics you're interested in graphing, you're ready to install jmxtrans.

How Do I Install It?

Jmxtrans is hosted on GitHub. You can download prepackaged versions for Red Hat or Debian systems, or a source code zip file from the downloads page [4]. The Red Hat and Debian packages both ship with an init script, which makes it easy to start or stop the jmxtrans service on those systems, and a config file in /etc that you can use to modify the behavior of the jmxtrans daemon itself (including its polling interval).

The jmxtrans service reads from configuration files placed in /var/lib/jmxtrans. Here's an example of a JSON config for

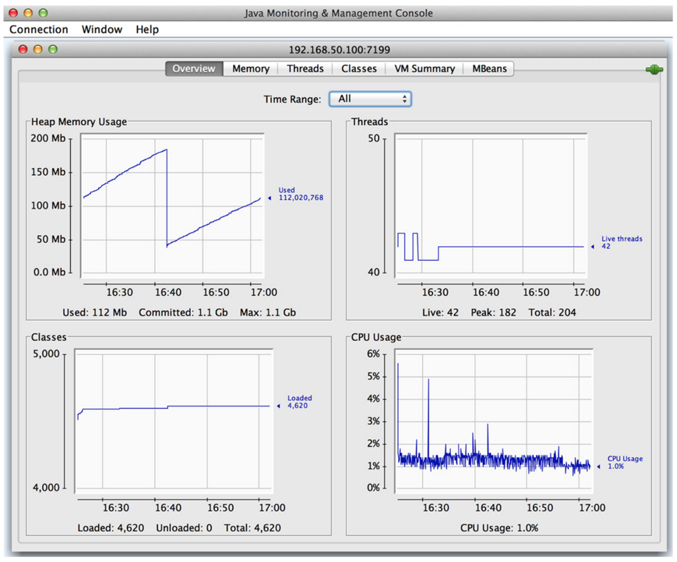

Java Monitoring & Management Console
Connection Window Help
192.168.50.100:7199
Overview Memory Threads Classes VM Summary MBeans
Time Range: All
Heap Memory Usage
Used 112,020,768
Used: 112 Mb Committed: 1.1 Gb Max: 1.1 Gb
Threads
Live threads 42
Live: 42 Peak: 182 Total: 204
Classes
Loaded 4,620
Loaded: 4,620 Unloaded: 0 Total: 4,620
CPU Usage
CPU Usage 1.0%
CPU Usage: 1.0%

Figure 3: JMX automatically exports CPU, thread, and memory for every application that runs on the JVM.

jmxtrans that'll capture a couple of Cassandra metrics that refer to compactions on a single Cassandra node, and send them to Librato.

```
{
  "servers" : [ {
    "host" : "192.168.50.100",
    "port" : "7199",
    "queries" : [ {
      "obj" : "org.apache.cassandra.
db:type=CompactionManager,*",
      "attr" : [ "PendingTasks", "TotalBytesCompacted"],
      "outputWriters" : [ {
        "@class" : "com.googlecode.jmxtrans.model.output.
LibratoWriter",
        "settings" : {
            "username" : "dave@librato.com",
            "token" :
"cd4b234567545cb2453243qcbt546dbd43d5371"
        }
      } ]
    } ]
  } ]
}
```

Jmxtrans uses *output writers* to emit metric data to a specific monitoring system. In the example above, we're using the Librato output writer, but many are available to push metrics to systems like StatsD, Graphite, RRDtool, and flat-files. Since the outputWriters attribute is a JSON array, you can specify more than one output writer for a given series of metrics, and jmxtrans will emit to all of them simultaneously. You should start to see data in your interface of choice in a few seconds (Figure 4).

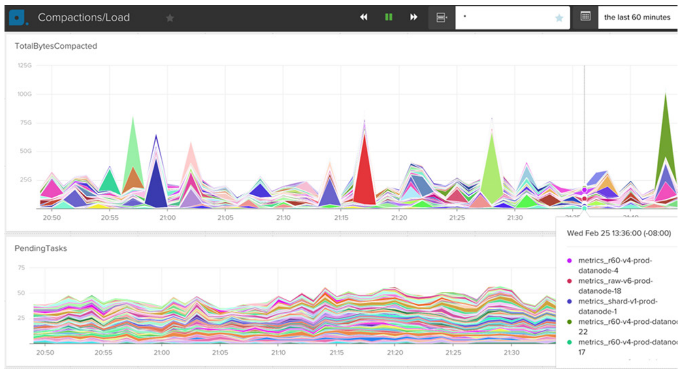

Compactions/Load
the last 60 minutes
TotalBytesCompacted
PendingTasks
Wed Feb 25 13:36:00 (-08:00)
metrics_r60-v4-prod-datanode-4
metrics_raw-v6-prod-datanode-18
metrics_shard-v1-prod-datanode-1
metrics_r60-v4-prod-datanode-22
metrics_r60-v4-prod-datanode-17

Figure 4: Data should appear in your metrics interface within 30 seconds.

Jmxtrans will also emit stack-traces and error logs into a log file in /var/log/jmxtrans. Check this file if you don't begin to see metrics appear in a few seconds.

Caveat Emptor

At the time of this writing, the jmxtrans project was in the process of moving their build process from Ant to Maven, and, as a result, the Librato output writer was not included in the packaged versions of jmxtrans. If you're installing jmxtrans from one of the pre-packaged distributions, and the Librato writer is not available, you can work around it by cloning this [5] jmxtrans repository, and building and installing it manually with:

```
maven clean install -Pdpkg
```

This notion I've been riffing on, that compelling software inevitably begets community, implies another way of looking at glue-code like jmxtrans in general. These tools, the ones that everyone uses but none of which will ever have anything like a conference of its own, form the substrate that joins communities together. As someone who has spent many years working on glue code, that's a comforting thought.

Take it easy.

References

[1] JVM Language Summit: http://openjdk.java.net/projects/mlvm/jvmlangsummit/.

[2] List of JVM Languages: http://en.wikipedia.org/wiki/List_of_JVM_languages.

[3] Secure installation of jmxtrans: http://docs.oracle.com/javase/8/docs/technotes/guides/management/agent.html.

[4] Jmxtrans download page: https://github.com/jmxtrans/jmxtrans/downloads.

[5] Jmxtrans developer fork: https://github.com/praste/jmxtrans/tree/dpkg.

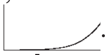
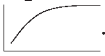
For Good Measure

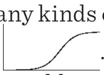
Much Ado about Metadata

DAN GEER AND DAN CONWAY

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

Dan Conway is Director of the Center for Business Analytics at Loras College. He has previously served on the faculty at Notre Dame, Indiana University, and Northwestern University. datasciencedan@gmail.com

Trendline metadata inform high frequency trading algorithms, advertising algorithms, and bandwidth balancing algorithms. Trendline metadata inform mitigation choices, budget priorities, and policy. When a trendline measures cumulative events, the shape is called "convex" if the underlying data is increasing in frequency . Similarly, the trendline's shape is called "concave" if the underlying data is decreasing in frequency .

The cumulative life of many kinds of adoption processes take an "s-curve" shape—convex at first and then concave . The s-curve pattern occurs everywhere: it describes the number of VAX computers sold in the 1980s, the prevalence of Internet access in Nigeria, the number of English articles posted on Wikipedia, the spread of cancer, and the adoption of just about any new technology. Parameterized s-curves pinpoint the "inflection point" where acceleration of the growth curve becomes zero and convexity converts to concavity. A parameterized s-curve estimates the final lifetime number of accumulated events as well. (There are lots of references on this if you want to learn more; "logistic" is a good search word with which to begin.)

We were curious about trends in categories of cybersecurity study and whether s-curves might be what we see there. The editors at *IEEE Security & Privacy* very kindly provided us with all keywords from 12 years of articles in *S&P*. Those keywords are certainly varied; there were 7501 keywords—3071 of them unique—spread across 1341 articles. As one might expect, the terms "security" and "privacy" were the most frequently used keywords with 1778 combined occurrences.

The cumulative density of all keywords is shown in Figure 1, indicating a slight increase in the number of keywords used.

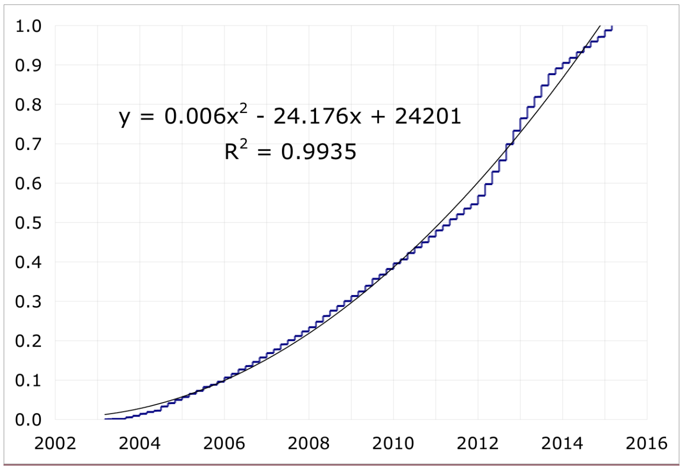


Figure 1: All keywords

Again, a curve that is not only increasing but also increasing at an increasing rate (even if only a slight acceleration) is called "convex." We will use Figure 1 as our basis of comparison to other curves you will see below, and, per convention, we will use twice the coefficient of the x^2 term, 0.006 x 2 = 0.012, as the "convexity" of the curve. The reason we are looking at this is simple: if a given keyword has a convex trendline, then it can be called a "leading indicator" and can be said to be predictive (in context). If the coefficient is negative and therefore the curve's upward growth is decelerating, then it is a "lagging indicator" and can be said to be descriptive.

So how do trendlines in specific keywords compare? If we consider the ten domains of the CISSP Common Body of Knowledge [1], we find a statistically similar pattern to Figure 1 in keyword alignment. The distribution of the domains differs, as "Network Security" appears 10x as often as "Physical Environmental Security," but the overall trendline shapes are identical. They are neither lagging nor leading. We will have to look beyond those domains to find different shapes that tell us more.

Take the keyword "crime"; it has a convex pattern, and Figure 2 tells us that authors' interest in "crime" is increasing (with coefficient .0330).

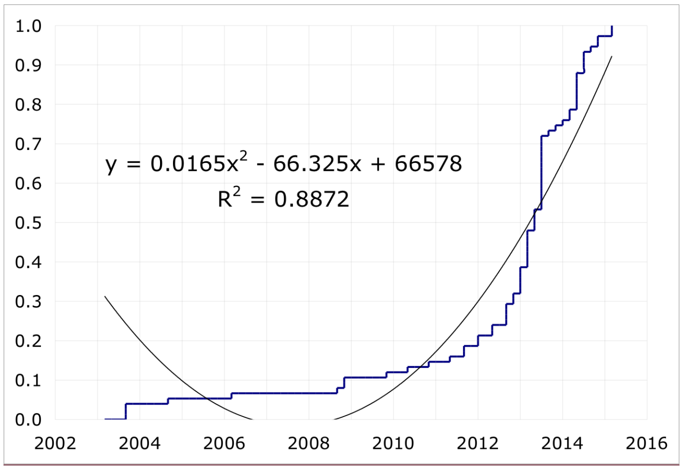


Figure 2: Keyword "crime"

The keyword "honey" (in several merged variations) follows a concave pattern with coefficient –.0017. It is a term that is no longer active in the keyword population, as seen in Figure 3.

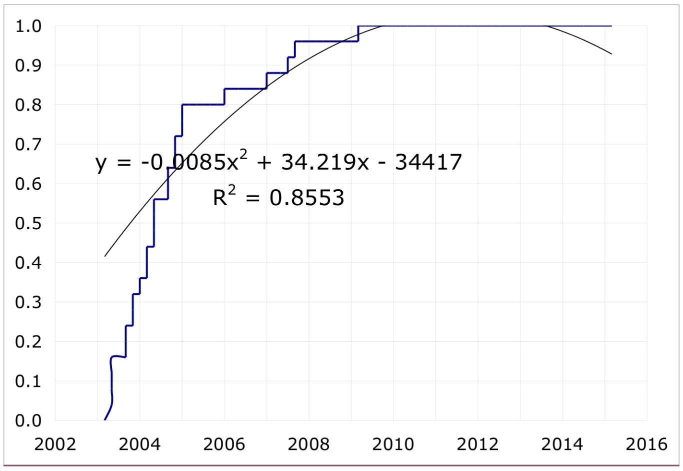


Figure 3: Keyword "honey"

"Virus" (Figure 4) follows a similar pattern to "honey." With a coefficient of –.0072, it has begun to fade away.

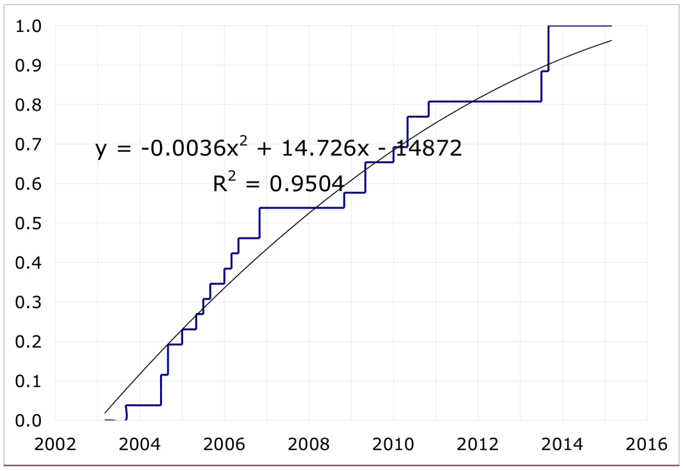


Figure 4: Keyword "virus"

Consider the keyword "identity" as shown in Figure 5. It has been used 56 times in the past 12 years, most recently in March of 2014. This term may be past its peak as indicated by its –.0042 coefficient.

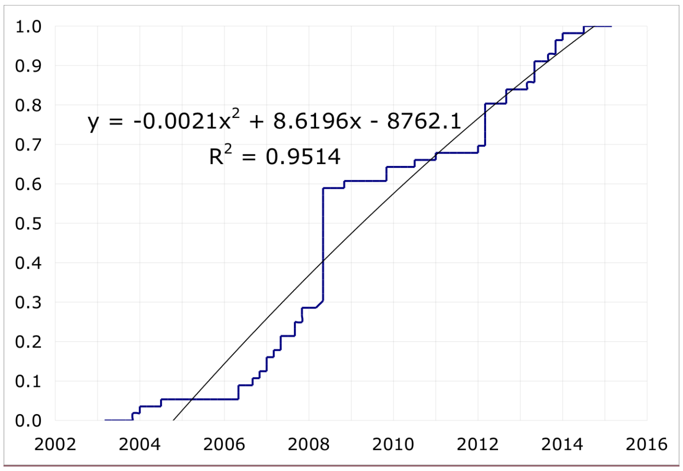


Figure 5: Keyword "identity"

Do these keyword trends reflect actual incidents involving identity? Stipulating that an article marked with the keyword "identity" is not necessarily an article about the misuse of identity information, we nevertheless went to the "dataloss database" [2] and plotted incidents involving loss of any two or more of ACC (account information), ADD (address information), CCN (credit card number), DOB (date of birth), MED (medical information), NAA (name), and SSN (social security number) over the same time period as the *S&P* keyword "identity." For the DatalossDB, we find the nearly linear pattern seen in Figure 6. Not much evidence here of the "identity" keyword being a leading indicator.

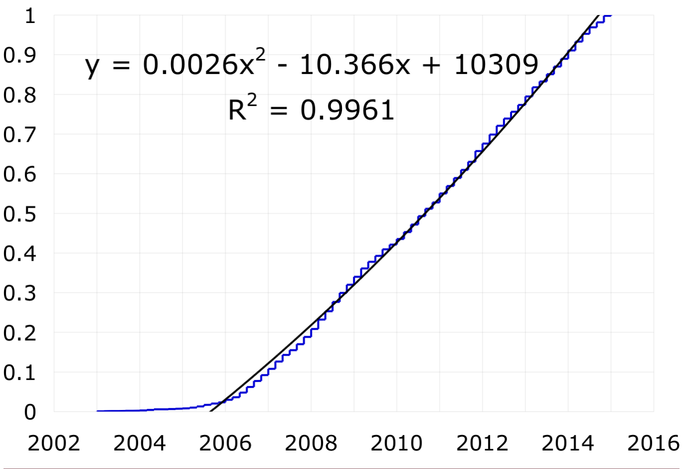


Figure 6: DatalossDB curve for "identity"

(As a side note, Symantec points out [3] that identity theft protection costs approximately $150/year whereas Personally Identifying Information (PII) is available in underground markets for $12–$16 each; hence PII is worth roughly ten times as much to the person identified as to external interests.)

Turning to "crime," we took as our real life measure the incidents reported to DatalossDB as "hack," "stolen [items]," and/or "fraud"—yielding the trendline seen in Figure 7. Compare the convexity coefficient .0186 here for this one measure of actual cybercrime to the .0330 for articles with the keyword "crime" as seen in Figure 2.

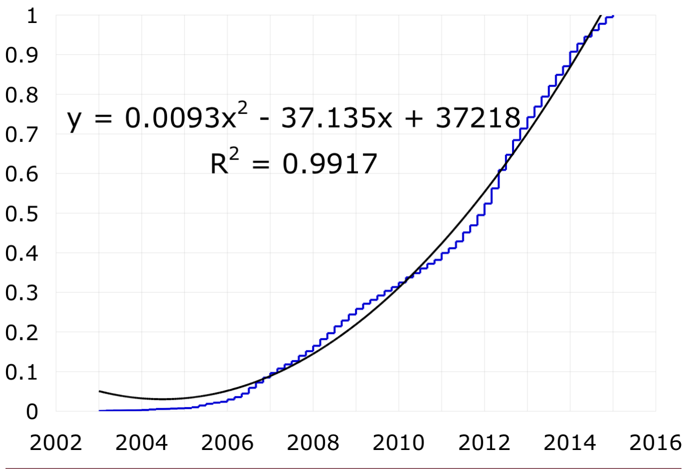


Figure 7: DatalossDB curve for "crime"

In marketing, when a new product is being evaluated, the company will estimate the number and timing of lifetime customers. Customers coming and disappearing soon thereafter are referred to as "churn," and churn models attempt to describe when members will leave a population, be it a population of cell phone customers, those with cancer, or members of an online forum. For example, HP and Cisco measure employee time on LinkedIn and such, attempting to determine who is likely to leave. For those they want to keep, they then intervene. A retailer will want to know the probability of a customer not returning, and send them a coupon to extend the customer lifetime.

If we treat keywords as customers of *IEEE S&P*, then one might ask when "crime" will peak as a customer and begin to decline. This forecasting technique uses the s-curve, which, by definition, is convex up to its "inflection point" and concave after that point. The keyword "crime" appeared 75 times across those 1341 articles. Solving a least squares fit of our s-curve to use of the keyword "crime," we get Figure 8. In other words, we might be now seeing the keyword "crime" start to appear less and less, and we might predict a total of 133 occurrences during its product lifetime (that is to say, before it isn't used anymore at all and the cumulative frequency curve asymptotes).

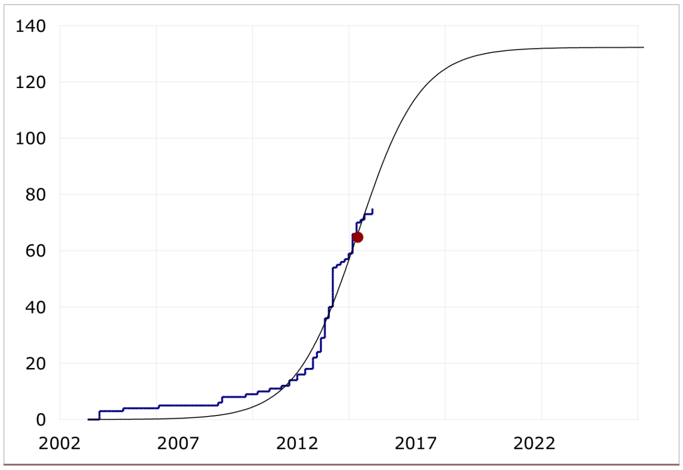


Figure 8: S-curve fit to "crime" keyword, inflection point on July 22, 2014

The term "virus" appears to have hit its inflection point in 2007 and is in active decline. We anticipate very few additional articles on this topic (see Figure 9).

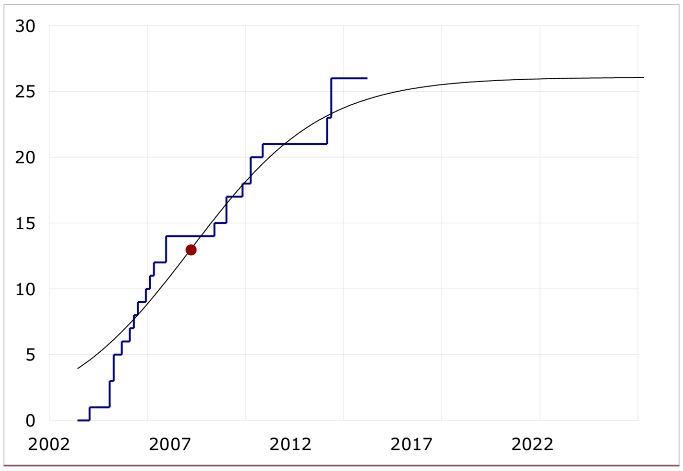


Figure 9: S-curve fit to "virus" keyword, inflection point on November 14, 2007

Let's try a newer term, like "cloud." It certainly came on strong, but is keyword use as shown in Figure 10 telling us that cloud security is approaching a solved problem?

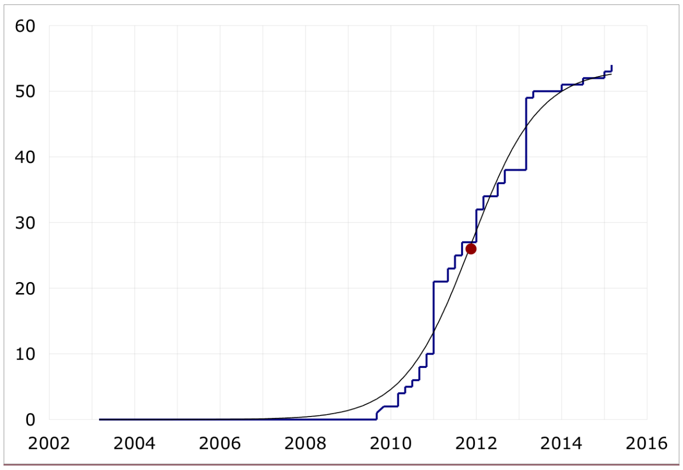


Figure 10: S-curve fit to "cloud" keyword, inflection point on November 17, 2011

Perhaps the keyword "ethic" is your interest, then see Figure 11.

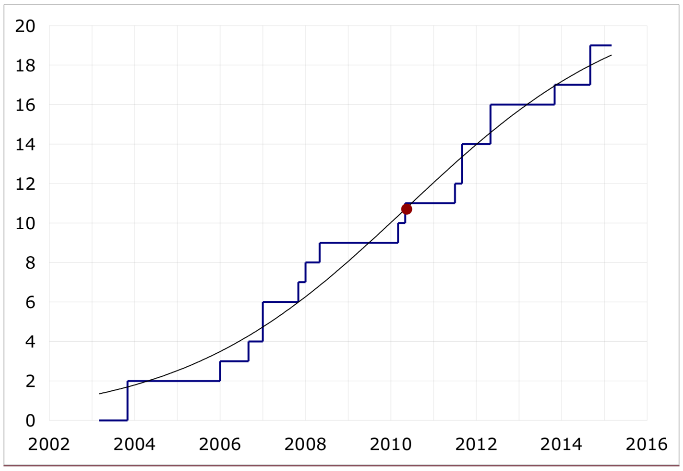


Figure 11: S-curve fit to "ethic" keyword, inflection point on May 16, 2010

The market for solutions seems to follow this lifecycle as well. Virus protection is now available for free, and identity management solutions are decreasing in price. And both terms are past their peak as keywords in *S&P*. Cloud computing adoption rates are clearly not being blocked by security concerns. "Crime" is still trending upward, so can we infer that spending on crime prevention and related services will continue for several years before peaking and beginning its own contraction.

Of course, the keywords chosen by authors of articles are not subject to any particular consistency control. As we said at the outset, there were 3071 distinct keywords across 1341 articles, guaranteeing a lot of singletons (2062 to be precise). We tried binning the keywords, getting far enough to end up with Table 1.

history	8
file types	18
controls	38
meetings	42
security and privacy	54
press	57
targets	71
roles	75
person	135
education	145
metrics	161
countermeasures	171
networks	173
analysis	202
cryptography	255
access control	266
privacy	397
policy	523
attack methods	844
security	1647
<other>	2219

Table 1: Binned keywords

When we binned them, we didn't actually see curves very different from direct use of this or that keyword by itself except for one case: when an author used "security and privacy" as a unitary keyword, rather than "security" and "privacy" as separate keywords, we did get an interesting graph (see Figure 12). Perhaps Figure 12 has something to say about whether "security and privacy" are an indivisible social good or two diverging ones.

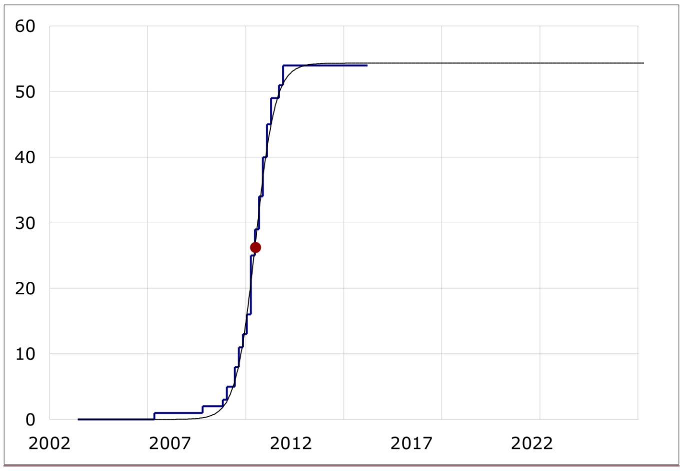


Figure 12: S-curve fit to "security and privacy" keyword, inflection point on July 13, 2010

There's a lot more to explore; we'll be back.

And thanks to you, again, IEEE and DatalossDB.org.

References

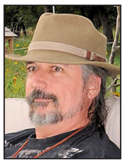
[1] https://isc2.org/cissp-domains/default.aspx.

[2] http://datalossdb.org.

[3] http://www.idtheftcenter.org/Privacy-Issues/how-much-is-your-identity-worth-on-the-black-market.html.

/dev/random
History, or Maybe Not

ROBERT G. FERRELL



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

I have decided this time to touch on history, specifically, that of USENIX. As such I've delved deeply into the musty dusty archives of my own frontal lobes, which are easier to get into than the, um, backal ones (why isn't this a word?) because most of the synapses in my midbrain have been co-opted by snatches of '70s rock songs, Monty Python quotes, and obscure factoids I got from Snapple Peach Tea lids. That makes accessing anything further aft problematic.

I have a peculiar, albeit quite minor, connection with USENIX history. It would be more accurate to say, rather, that I have a distinctive non-connection to it, to wit: I served, or failed to serve, as the historian for SAGE. The story behind this confusing sea of negation begins with a notification to SAGE members via *Sys Admin* magazine in late 1997 or early 1998.

The notice was a solicitation of candidates for the newly created office of historian for SAGE, the somewhat puzzling acronym for USENIX's Systems Administrator's Guild(e). I had served as historian for a couple different chapters of a nonprofit, so I figured I'd give it a shot. Apparently I and another gentleman were the only ones to apply, so the powers that be decided we could share the title of SAGE Historian. We contacted one another and decided on the approach we would take to this most engaging assignment, which mostly consisted of each doing their own thing without any further communication.

While there are many luxuries a burgeoning co-historian can eschew and still function at an acceptable level, there is one commodity without which they are, to employ the vulgar vernacular, screwed: history itself. Alas, that was the one thing we (I) could not seem to generate no matter how hard we (I) tried.

It isn't as though there was no history to be had; the very fact that USENIX leadership saw fit to appoint a brace of historians for SAGE strongly suggests otherwise. No, the problem we (I) experienced centered chiefly on our (my) profound inability to coax meaningful bits of that history from members who lived it. I wrote letters, made phone calls, cajoled, wheedled, and flattered my way to…not much history. I got a plethora of promises; my inbox was filled with good intentions and best wishes—but not a lot of anything useful.

So I chose another approach that had served me well in past assignments: I decided to write my own wholly fictional history and then send it out, hoping the readers would take umbrage at my fabrications and correct them. It didn't work. Some of them even told me they remembered this or that incident and thought I recounted it rather well. It dawned on me after a few weeks of this that SAGE history might be easier simply to invent than investigate. I wondered how often this principle had occurred to previous chroniclers of the human condition. It would explain a lot.

The historian position and I drifted apart after a few months; my "history" was never published or otherwise promulgated, until now. I have excavated and below present for your edification and amusement selected excerpts from my larger *History of UNIX and USENIX, as Told by Some Guy Sitting Next to Me on the Subway.* It is, as I've indicated, completely and unashamedly fictitious, with no regard whatever to truth, the historical record, or even basic physical laws.

UNIX began as an improved system for generating man pages. The old method, using a *man*ual typewriter, a ball-peen hammer, and five-to-seven wads of well-chewed bubblegum, was tedious and slow, prompting Dennis Ritchie and Ken Thompson to develop a better mechanism in their spare time. They began by rewriting the premises for *Have Gun–Will Travel, Mission: Impossible,* and *Hogan's Heroes* in Sanskrit, then converting that to PL/I using semaphore flags borrowed from local Boy Scouts.

Once the more serious lacerations had been stitched and the cracked windowpanes replaced, they punched the resulting characters into cards and put them on the mainframe in batch mode for the night. The next morning they discovered that they'd created an entirely new operating system which they decided, after staring for too long at the computer lab aquarium, to call "Xinu" because it sounded like they imagined a blue neon guppy would sneezing. They wrote the name on the only surface they could find, the aquarium wall. That night the cleaning lady wiped it off, but not before copying it down from the other side of the tank in case it was important—and thus was UNIX born. (She had left school to enter the janitorial field before they learned the lowercase letters.)

People began to employ this new operating system because the others available at the time were too easy to understand, and after a few years those early adopters banded together to form a users' group and bowling team. Searching for a name, they first wanted the cleverly named *UNIX Users' Group* but that did not fit on the sweat bands very well. They then tried *Users of UNIX,* but the guy who was charged with etching this name into the beer mugs only had enough etch for six (capital) letters, so he shortened it to USENIX.

USENIX attracted only 40 people or so at first, many of whom thought they were attending the Undergarment Sales Executives' Northern Indiana eXposition, but gradually membership expanded to include people who knew the difference between a dumb terminal and a bus terminal; after this the organization really got rolling.

At some point the steadily expanding membership decided to start sponsoring technical conferences. The natural first step in this process was to appoint a committee to come up with cool acronyms based on irrelevant adjectives, cutesy names, and obscure mythological creatures. After that crucial benchmark had been cleared it was simply a matter of choosing topics, arranging speakers and venues, and, well, all the other stuff that goes into giving technical folks an excuse to duck out of the keynote and explore the drinking establishments in a distant city.

I suppose this isn't really much of a history, but I only have a limited space here and, as I noted above, my call for real data wasn't so much answered as aggressively ignored. Sure, there's Wikipedia, but where's the fun in that? At least I admit from the outset that I'm shoveling manure at you.

Book Reviews

MARK LAMOURINE

Graph Databases
Ian Robinson, Jim Webber, and Emil Eifrem
Neo Technology Inc., O'Reilly Media, 2013, 210 pages
ISBN 978-1-449-35626-2

Graph Databases starts off really well. It opens with a definition and then a detailed description of what makes up a graph database and what makes graph databases special.

Some of it is pretty striking. I remember the "aha!" moment when I read that conventional relational databases don't contain relationships as first-class entities. It's true. Relational databases force all data into a table form, and then relationships are represented using second-class column types (FOREIGN KEY) and JOIN operations. In a graph database the relationships between data entities are first-class entities of their own.

The authors present ways of representing data relationships both graphically (naturally enough) and textually, as is required to communicate with a service over network (serialized) media. It's when they get to demonstration and implementation that I started to lose my enthusiasm.

While the title might lead a reader to believe that the book covers graph databases in general, it turns out that only one database system is presented. In the middle of Chapter 4, the authors begin talking about implementation options, but the only real option offered is Neo4j. I continued reading on, hoping to find some variation or alternatives, something to lead back to the general topic of graph databases, and alternate implementations, but I was disappointed. At some point I stopped and looked up two different things:

1. Graph database implementations—there are dozens
2. The authors—all employees of Neo4j

In fact, the copyright for the book is held by Neo4j Inc., not by the authors. To be fair, the forward and the bios on the back cover both make it clear that the three authors are the co-founders of Neo4j. If the title of the book had been "Graph Databases with Neo4j" I would not have felt so disappointed, but then I might not have picked it up in the first place. Once I realigned my expectations, I went back to reading.

In the remainder of the book, the authors talk about real-world instances of data well suited to graph modeling and representation in a graph database. Many applications today represent the links between people, objects, and concepts in a mesh or network (in the mathematical or social sense). Whether the application is an enterprise network or a social one, the links between nodes are more important than the individual attributes when trying to discover patterns of behavior or data flow.

Near the end of the book, the authors discuss database internals with an eye to performance, reliability, and scaling. They close with a chapter on how common problems can be represented more intuitively with a graph than with more traditional data structures.

Overall, the writing is clear, and the progression led me to a better understanding of how a graph database works. I will definitely be more likely to recognize an application that would suit modeling and implementation with a graph database, and I would certainly consider Neo4j for the implementation. I do wish that the focus on Neo4j had been more explicit in the title and on the front cover. If you want to learn about Neo4j, this is certainly a good place to begin.

Interactive Data Visualization for the Web
Scott Murray
O'Reilly Media Inc., 2013, 256 pages
ISBN 978-1-449-33973-9

I spend a fair amount of time drawing boxes and lines when I'm trying to explain things in documentation. Often I'm representing tabular data or collections with relationships. When it comes to representing data sets that change over time, I generally don't even try.

Recently, though, I had a problem I needed to understand myself: What are the relationships between RPMs with dependencies? And what effect does the addition of a single package with complex dependencies have on the total package set installed on a host? I didn't want to see just the list of new packages to add but where (and why) each new package was pulled in. I also wanted to be able to view what would change if I added just a subset or tried to remove a dependency. I remembered that a coworker had created some great dynamic visualizations using reveal.js, but that by itself didn't seem to have the data representation I was looking for. I asked him what he used. He pointed me at D3, and I went looking for books.

D3 is a JavaScript library (available at http://d3js.org). You embed it in your application in the same way you would JQuery or any other JavaScript library. D3 offers me a capability for browser-based graphics and data visualization that I just would have found impossible otherwise, and Scott Murray's book was a great way to get started.

Murray starts off with the traditional definition and features, but he's careful to outline what D3 is and is not good for. He devotes an entire section to other tools that might suit better than D3 depending on your needs and your application. He spends another large section bringing readers new to Web programming for browsers up to speed. There's a short section on HTML and

DOM programming as well as simple SVG and HTML canvas programming. This is sprinkled with outside resources so readers can get more detail and come back if needed.

This is when Murray really gets rolling. He shows how to use D3 to retrieve your data from a Web server and how to use it to populate your document with HTML and SVG elements, which the browser will draw for you. Simple graphing such as scatterplots and bar charts requires some labeling and scales for the axes, and each of these gets a section, as do dynamic updates and user interaction.

Things get really cool when Murray gets to D3 layouts. Layouts provide ways for D3 to automatically place the elements and draw them and even move them around in response to clicks and drags in the browser. When using layouts, you don't specify where each data element will be placed. Rather, D3 does it for you dynamically based on the data values themselves.

Murray demonstrates three common layouts. The Pie and Stack layouts yield fairly common-looking graphics. The Force layout, Murray admits, is overused because it is so cool. In a Force layout each data object is assigned a repulsive force. The elements are arranged randomly at first (constrained by links between elements that are related in some way), and the Force layout applies the forces, moving each element until they reach equilibrium. This is very slick to watch and it is seductive. I used the Force layout for my RPM dependency data, but a Tree layout might have been more appropriate, and I am going to try it to see.

The layout and writing of Interactive Data Visualization are themselves appealing. This is one of the first books of this type that I've read which uses full color for both the illustrations and for the example texts. The source code and HTML representations are taken from the Safari browser debugging tool set, and the colorization of the text is a welcome and familiar feature (although I use Emacs and Chrome).

As you may have noticed, I'm pretty enthusiastic about both D3 and Interactive Data Visualization. As a sysadmin with a strong coding background and some experience with JavaScript and browser development, I have with D3 a powerful new tool for understanding and explaining the behavior of the systems I'm working on using output from the CLI tools I already have, but it's unlikely that I'll use it often enough to stay fluent. Murray's book is one I'll return to for a refresher when I find a new question that cries out for a graphic representation.

Scratch Programming in Easy Steps
Sean McManus
In Easy Steps Ltd., 2014, 216 pages
ISBN 978-1-84078-1
I've looked at Scratch and reviewed books on it before, but something made me pull Scratch Programming off the shelf at my local bookstore and thumb through it. I was impressed instantly with its high-quality feel, the texture of the paper, and the weight of it in my hand. It's not a thick volume but its heft makes me think it's likely to be durable in the hands of the middle school students I think it is meant for.

Once I started working through the book, I noticed something else right away: Scratch has become a Web application. Scratch 2.0 is hosted by MIT at http://scratch.mit.edu. You can open the development screen in any modern browser. If you create an account and log in, you can save and publish your applications. Scratch 1.4 is still available as a standalone application. Scratch Programming uses Scratch 2.0 as its base, but it also includes information on running Scratch 1.4 on a Raspberry Pi. The text consistently and clearly includes graphics and instructions that show how 1.4 will differ from the default.

Getting and keeping the interest of middle schoolers can be a challenge. In books like this, I look for the hooks that will help hold the attention and enthusiasm of the students. McManus does a great job of mixing narrative with engaging graphics and layout to sustain interest.

Scratch Programming is color coded to make it easy to find one's place and return quickly to work when picking the book up, and each chapter presents a project, a mixture of simple graphical games, musical applications for sound, quiz games, and logic puzzles. Each section brings a new aspect of coding and makes good use of the Scratch graphical programming model to illustrate the points. McManus leaves enough room for the kids to experiment, make mistakes, and discover the solutions for themselves. He builds each concept or construct in a linear way, allowing for the reader to race ahead or off on a tangent and be guided gently back.

The last two sections bring in discussions of hardware sensors that can be used to provide additional inputs for game behavior and response. McManus covers using a computer's Webcam or a USB device called a "Picoboard," which can interface with Scratch to respond to sounds (such as a handclap) and changes in light or temperature.

In the final chapter, McManus provides seven short, complete programs and encourages the reader to experiment with them, changing parameters or logic and observing how the changes affect the behavior of the program.

I'm impressed with *Scratch Programming*, and I actually followed several of the projects to completion in Scratch because I was having fun in a way I hadn't since I'd done similar things on the Apple in my own high school. I have a couple of friends with girls the right age who've expressed interest in coding, and I mean to pass on the review copy and maybe even buy an additional one to give to them.

USENIX Board of Directors

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

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

Vice President
John Arrasjid, *EMC*
johna@usenix.org

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

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

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

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

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

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

Executive Director
Casey Henderson
casey@usenix.org

USENIX Member Benefits

Members of the USENIX Association receive the following benefits:

- **Free subscription** to *;login:*, the Association's magazine, published six times a year, featuring technical articles, system administration articles, tips and techniques, practical columns on such topics as security, Perl, networks, and operating systems, book reviews, and reports of sessions at USENIX conferences.
- **Access** to *;login:* online from December 1997 to the current month: www.usenix.org/publications/login/
- **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

Conference Reports

In this issue:

13th USENIX Conference on File and Storage Technologies

February 16–19, 2015, Santa Clara, CA

Summarized by Zhen Cao, Akhilesh Chaganti, Ming Chen, and Rik Farrow

Opening Remarks

Summarized by Rik Farrow

Erez Zadok opened the conference with the numbers: 130 submissions with 28 papers selected. They used a two round online review process, with each paper getting three reviews during the first round, and the 68 remaining papers getting two more reviews in the next round. The final decisions were made in an all-day meeting at Stony Brook.

Jiri Schindler announced awards, starting with ACM Test-of-Time Fast Track awards going to "RAIDShield: Characterizing, Monitoring, and Proactively Protecting against Disk Failures," by Ao Ma et al., and "BetrFS: A Right-Optimized Write-Optimized File System," by William Jannen et al. The Best Paper Award went to "Skylight—A Window on Shingled Disk Operation," by Abutalib Aghayev and Peter Desnoyers. These researchers cut a window into a shingled (SMR) drive so that they could use a high-speed camera to record disk seeks, an interesting form of reverse engineering. I preferred the BetrFS paper myself, and asked the authors to write for *;login:* about the B-epsilon trees used to speed up writes. But, not surprisingly, I wasn't part of the PC, which did the work of selecting the 21% of submitted papers.

The Test-of-Time award, based on papers published between 2002 and 2005 at FAST, went to "Hippodrome: Running Circles around Storage Administration," by Eric Anderson et al. Hippodrome is a tool used to automate the design and configuration process for storage systems using an iterative loop that continues until it finds a satisfactory solution to the target workload. Most of the authors, all HP Lab employees, were present to receive the award.

Erez Zadok wanted his students to have the experience of summarizing presentations. Erez also reviewed their summaries before the students sent them to me for editing.

The Theory of Everything: Scaling for Future Systems

Summarized by Ming Chen (mchen@cs.stonybrook.edu)

CalvinFS: Consistent WAN Replication and Scalable Metadata Management for Distributed File Systems

Alexander Thomson, Google; Daniel J. Abadi, Yale University

Alexander Thomson began by noting that their distributed file system was the first attempt to solve two difficult problems simultaneously: consistent file system replication over wide-area network (WAN), and scalable metadata management. He emphasized the importance of the problems and reviewed previous systems (HDFS, Ceph, etc.) which each addressed one of these problems but not both. He then went through a file creation example to demonstrate how CalvinFS handles file system operations and solves both problems.

CalvinFS achieves strong consistency across geographically distant replicas and high metadata scalability by using CalvinDB, a partitioned database system that supports fast distributed transactions. CalvinDB operates by (1) ordering and scheduling all transactions globally, (2) reading and broadcasting all transaction inputs, and then (3) deterministically committing the transactions on all replicas once all needed inputs and locks become available. CalvinDB's deterministic feature allows transactions to be committed without requiring a traditional two-phase commit protocol. Consequently, each CalvinDB transaction incurs only one network RTT latency instead of two as in two-phase commit. To be deterministic, the read- and write-set and locks of transactions need to be known in advance, which is not always possible: for example, when the write-set depends on the results of preceding read operations in the same transaction. CalvinDB solves this using an Optimistic Lock Location Prediction (OLLP) mechanism. In addition to the metadata store based on CalvinDB, CalvinFS also has a variable-size immutable block store, which associates each block with a global unique ID (GUID) and assigns it to block servers using hashing.

CalvinFS can store 3 billion small files that are replicated among three geo-distributed datacenters (each running 100 EC2 instances) and could store even more if the authors could afford to run more instances. CalvinFS can handle hundreds of thousands of updates and millions of reads per second. By satisfying reads locally, CalvinFS has low read latency (median of 5 ms, and 99th percentile of 120 ms); however, as a tradeoff for strong global consistency, its writes suffer from WAN latency (200–800 ms).

In the Q&A session, Alexander acknowledged Zhe Zhang's (Cloudera) supposition that directory renaming is more expensive than other recursive operations, such as changing permission of a directory. Zhe then asked about the order between writing the block data and creating the file in the example. Alexander answered that the order does not matter as long as the two subtasks are committed at the same time. Brian (Johns Trading) wondered how CalvinFS handles conflicting operations made simultaneously by multiple datacenters. Alexander replied that conflicting operations are easy to detect and handle because CalvinFS has a global ordering of all operations.

Analysis of the ECMWF Storage Landscape

Matthias Grawinkel, Lars Nagel, Markus Masker, Federico Padua, and Andre Brinkmann, Johannes-Gutenberg University Mainz; Lennart Sorth, European Centre for Medium-Range Weather Forecasts

Matthias presented the first analysis of active archives (data to be accessed at any time), specifically a 14.8 PB general-purpose file archive (ECFS) and a 37.9 PB object database (MARS) at the European Centre for Medium-Range Weather Forecasts (ECMWF). Both systems use tapes as primary storage media and disks as cache media.

As of September 2014, the ECFS archive system had a 1:43 disk-to-tape ratio, and it contained 137.5 million files and 5.5 million directories. From 1/1/2012 to 5/20/2014, ECFS served 78.3 million PUT requests (11.83 PB in total size) involving 66.2 million files; 38.5 million GET requests (7.24 PB in total size) involving 12.2 million files; 4.2 million DELETE requests; and 6.4 million RENAME requests. The requests demonstrated high locality, enabling an 86.7% disk cache hit ratio. Nevertheless, 73.7% of the files on tape have never been read. In the simulation study of various caching algorithms, the Belady algorithm performed the best, the ARC and LRU algorithms followed closely after, and MRU was the worst.

The MARS object database had a 1:38 disk-to-tape ratio, and contained 170 billion fields in 9.7 million files and 0.56 million directories. From 1/1/2010 to 2/27/2014, MARS served 115 million archive requests and 1.2 billion retrieve requests. MARS's cache was very effective, and only 2.2% of the requests needed to read data from tapes. Similar to ECFS, 80.4% tape files in MARS were never read.

Matthias also presented some interesting tape statistics in the HPSS backing ECFS and MARS: (1) there were about nine tape loads per minute; (2) about 20% of all tapes accounted for 80% of all mounts; (3) the median of the tape loading time was only 35 seconds, but its 95% and 99% percentiles were two and four minutes, respectively.

Matthias concluded that disk caches on top of tapes are efficient in non-interactive systems. He noted two drawbacks of heavy use of tapes: high wear-out and unpredictable stacking latencies. He also expressed one concern, considering that the decrease in per-bit storage cost is slowing, of accommodating the fast-growing storage requirement (up to 53% annually) under a constant budget. Their traces and scripts are published at https://github.com/zdvresearch/fast15-paper-extras in order to help build better archive systems.

Umesh Maheshwari (Nimble Storage) wondered what the cost effect would be if the data were stored purely in disks since disk price had dropped and disks could eliminate the difficulties of loading tapes. Matthias thought pure disks would still be more expensive, especially considering that disks keep spinning (and consuming energy). Brent Welch (Google) commented that it should be possible to simulate how much more disk cache would be needed for the tape-and-disk hybrid system to perform as well as the more expensive pure-disk solution. John Kaitschuck (Seagate) wondered whether the data reported in this study contained traffic introduced by data migration (e.g., from an old format to a new format). Matthias was not sure of the answer.

Efficient Intra-Operating System Protection against Harmful DMAs

Moshe Malka, Nadav Amit, and Dan Tsafrir, Technion–Israel Institute of Technology

Moshe presented their work on improving the efficiency of using IOMMU (Input/Output Memory Management Unit), whose relationship to I/O devices is similar to the regular MMU's relationship to processes. IOMMU manages direct memory accesses (DMA) from I/O bus to main memory by maintaining mappings from I/O virtual addresses to physical memory addresses. Similar to MMU's TLB cache, there is an IOTLB cache for IOMMU. To protect the OS from errant/malicious devices and buggy drivers, IOMMU mappings should be established only for the moment they are used and be invalidated immediately after the DMA transfers. However, the IOTLB invalidation was slow and the IOVA (I/O Virtual Address) subsystem opts for batching (deferring) multiple invalidations.

Using 40 Gbps NICs as examples, Moshe showed from benchmarking data that the slowness of the invalidation was actually mainly contributed by the IOVA allocator software instead of the IOTLB hardware. The IOVA allocator used a constant-complexity algorithm based on a red-black tree and a heuristic.

The allocator works well if the NIC has one transfer ring and one receive ring. However, modern NICs usually have many transfer and receive rings, interaction among which can break the heuristic and convert the algorithm's complexity to linear. To solve the problem, Moshe showed how they converted the algorithm back to constant-complexity by simply adding a free list before the red-black tree. Through comprehensive evaluations, Moshe showed they improved the IOVA allocator by orders of magnitude and improved workloads' overall performance by up to 5.5x. They performed their study on Linux, but they found a similar problem on FreeBSD.

Raju Rangaswami (FIU) asked whether they had found similar problems in fast multi-queue SSDs, which likely suffer from the same interference among different queues. Moshe replied that another student in his lab was actively working on that and might come up with results in a few months.

Big: Big Systems

Summarized by Akhilesh Chaganti (akhilesh.chaganti@stonybrook.edu)

FlashGraph: Processing Billion-Node Graphs on an Array of Commodity SSDs

Da Zheng, Disa Mhembere, Randal Burns, Joshua Vogelstein, Carey E. Priebe, and Alexander S. Szalay, Johns Hopkins University

Da Zheng started the Big Systems session with their paper on large-scale graph analysis. He introduced FlashGraph, a semi-external memory-based graph analysis framework built over commodity flash arrays to achieve scalability and performance comparable to in-memory alternatives. He cited the application of graphs in many real world problems ubiquitously across industry and research as the main driver for developing cost-effective efficient alternatives like FlashGraph. Da Zheng explained the major challenges involved in graph analysis: (1) massive sizes with billions of nodes and edges, (2) the randomness of vertex connection, and (3) the power-law distribution in vertex degree, in which only a small number of vertices connect to many other vertices, making load balancing a problem in distributed environments. Da Zheng then presented the audience with the some popular alternatives employed in graph analysis and their problems. The first option is to use expensive machines with large RAM where the problem of cost and scalability is evident. Another alternative is to leverage a cluster to scale out graphs. Although this is popular, it is plagued by network latencies owing to the random vertex connections and the frequent network communications they invoke. The third option is to scale graph analysis using HDDs thanks to its capacity to store very large graphs. But random I/O performance and frequent I/O from external memory-based graph engines makes this solution very slow.

Da Zheng presented his position on how SSDs could offer a better solution in scaling graphs. SSDs have some obvious advantages compared to HDDs in terms of throughput and latency, while being cheaper and larger compared to RAM. But there are some potential problems that need to be addressed before using SSDs: heavy locking overhead for reads on large SSD arrays, and low throughput and high latency compared to RAM. Three design decisions are used to tackle these problems: reducing I/O, overlapping I/O and computation, and sequential I/O. All the desired choices are implemented using SAFS, a user space file system optimized for SSD array. Da Zheng then talked about the rest of the design, where FlashGraph communicates with SAFS and provides a graph-programming interface to the application layer. The main task of FlashGraph is to schedule the vertex programs written by the users and to optimize the I/O issues by those programs. When FlashGraph receives tasks, part of the computation is pushed to page cache to overlap computation and I/O and to reduce the memory overhead. The FlashGraph solution maintains the vertex state of the graph in memory while storing the edge lists on SSDs. The advantages with this model are quite evident. Compared to significant network-based communications in the distributed memory model, this model provides in-memory communication for all the vertices. This is better than the external memory model in terms of I/O. Even with SSDs, there is need for heavy I/O optimization. To achieve this, FlashGraph fetches only the lists required by the application. It also conservatively merges I/O requests on same or adjacent pages to increase sequential I/O.

Da Zheng presented the performance results of FlashGraph over some frequently used access patterns of graph applications. The classes of applications consisted of (1) those where only a subset of vertices access their own edge lists (e.g., Breadth First Search); (2) applications where every vertex accesses their own edge list (e.g., PageRank); and (3) applications where vertices access edge lists of other vertices (e.g., Triangle Count). The authors compared FlashGraph with other in-memory alternatives like PowerGraph (distributed memory) and Galois; to better understand its performance, they also benchmarked it against an in-memory implementation of FlashGraph. The in-memory implementation of FlashGraph is comparable to the performance of Galois, and the semi-external version of it is comparable to in-memory FlashGraph, sometimes reaching 80% of its performance. Surprisingly, FlashGraph outperforms PowerGraph in most cases. FlashGraph is also benchmarked for its scalability using the publicly available largest graph with 3.5 billion nodes and 129 billion edges. All classes of the algorithms run in a reasonable amount of time (from five minutes for BFS to 130 minutes for Triangle Count) and space (from 22 GB for BFS to 81 GB for Betweenness Centrality). The authors also used runtime results of Google Pregel and Microsoft Trinity on a smaller problem and better infrastructure to indirectly compare and conclude that FlashGraph outperforms them. Da Zheng concluded his talk saying that FlashGraph has performance comparable to in-memory counterparts and has provided opportunities to perform massive graph analysis on commodity hardware.

In the Q&A session, one participant asked why not use a NVME interface instead of using different file systems to improve read performance of SATA-based SSD and wondered whether reads would be compute-bound irrespective of the interface. Da Zheng replied that NVME will definitely improve performance because SAFS merges the I/O, which makes large and sequential reads overlap computation with I/O. Brad Morrey (HP Labs) asked whether adjacent pages in SSD were physically adjacent. Da Zheng replied they use the mapping table in SSD.

Host-Side Filesystem Journaling for Durable Shared Storage

Andromachi Hatzieleftheriou and Stergios V. Anastasiadis, University of Ioannina

Andromachi Hatzieleftheriou gave a lively presentation on improving the durability of shared data storage in a distributed file system environment in a typical datacenter-like clustered infrastructure. Before talking about durability problems, the author first outlined how typical datacenter storage is implemented as a multi-tier distributed system across clustered commodity servers and storage. In such a system, data is usually replicated on the client-side followed by a caching layer and the backend storage. The clients are implemented as stateless entities to reduce the communication across layers, which compromised the durability guarantees on the data present in the client's volatile memory in case of crash/reboot. The author has demonstrated the level of inconsistency such a model can cause over the Ceph object-based scale-out file system. While running Filebench fileserver workload over ten thousand files for two minutes, on average 24.3 MB of dirty data is present in the volatile memory because of write-back happening every five seconds over data older than 30 seconds. Network storage is usually implemented by providing one of either file-based, block-based, or object-based interfaces to clients. One important aspect of shared storage is the durable caching on the client side for performance and efficiency. Most approaches use block-based caching, which comes with significant problems like file sharing, block translation overhead, and consistency issues that pop up due to the semantic gap between file and block layers, which complicates the atomic grouping of dirty pages and the ordering of I/O requests.

The author then described how they improved durability of in-memory based caches on the host side using a file-based interface with better performance and efficiency. Key principles of their proposed storage architecture included a POSIX-like file interface with file sharing across different hosts, durability of recent writes in case of client crash/reboot, improved performance for write requests, and efficiently scaling out the backend storage. The author proposed Arion which uses these design principles and which relies on scale-out object-based backend system with multiple data (DS) and metadata servers (MDS). The clients (bare-metal or virtual) of the distributed file system are integrated with a disk-based journal that logs both data and metadata of I/O requests before the predefined numbers of backend replicas are created. Dirty data in memory is synchronously transferred to the journal either periodically or on explicit flush, and write-back to the backend server is initiated periodically or under tighter space conditions of memory or journal. Once write-back is complete, the corresponding entries in the journal are invalidated. The author then talked about handling the consistency issues during file access. Shared file access is achieved through the tokens leased to the client by MDS. Upon any concurrent conflicting operation by a different client, the client first checkpoints the pending updates, which is followed by invalidation of journal entries before revocation of token. Upon client reconnection or reboot, the client acquires the token again and replays file updates if journaled metadata is newer than metadata fetched from backend storage. For implementation, the Linux JBD2 is used to manage the journal, which is integrated with CephFS kernel-level client. In order to manage the file metadata in the journal, the JBD tag is expanded with fields to identify the metadata. This tag is used to compare the changes with metadata fetched from MDS.

The author provided the performance results of various experiments conducted on virtual clients over the Linux host with a storage backend cluster of five machines running Arion and Ceph. Arion was allocated a 2 GB partition for the local journal. All the experiments were based on Filebench and FIO workloads. Different configurations of Ceph and Arion were included in the benchmarks. Ceph was examined with write-back and expiration times set to 5 and 30 seconds (Ceph) and with both times set to 1 second (Ceph-1). Another configuration of Ceph was used where the file system was synchronously mounted (Ceph-sync) eliminating caching. Arion was set to copy changes to the journal every 1 second, while the write-back and expiration times were set to 60 seconds (Arion-60) or infinity (Arion-inf). The benchmarks were performed to analyze performance, durability, and efficiency of Arion compared to the Ceph kernel level client. Arion was found to reduce the average amount vulnerable data in memory to 5.4 MB for Filebench fileserver workload, significantly outperforming the Ceph client (24.3 MB). In another experiment over the Filebench Mailserver workload, Arion achieved up to 58% higher throughput than Ceph. Network traffic was also reduced by 30% in received load and 27% in transmitted load at one of the OSDs of the Ceph backend. Both the systems were also explored with the FIO microbenchmark with Zipfian random writes. The Arion-60 configuration achieved 22% reduced write latency compared to the default Ceph configuration, and total network traffic was reduced by 42% at one OSD. In the same experiment, the authors benchmarked bandwidth utilization of the file system of OSD and interestingly found that Arion reduced total file system disk utilization by 82%. Andromachi concluded her talk by giving the direction to future work and reiterated the benefits of host-side file journaling.

In Q&A, one participant wondered why the Ceph-sync configuration, where the file system is mounted with sync, had better throughput and network load. Another participant applauded the work and wondered how Arion handles a data loss case where the client commits to the local journal and goes down and meanwhile another client updates the corresponding file in the backend storage.

LADS: Optimizing Data Transfers Using Layout-Aware Data Scheduling

Youngjae Kim, Scott Atchley, Geoffroy R. Vallee, and Galen M. Shipman, Oak Ridge National Laboratory

Youngjae Kim began by giving the background of big data trends across scientific domains. The major challenges are storing, retrieving, and managing such extreme-scale data. He estimated that the Department of Energy's (DOE) data generation rates could reach 1 exabyte per year by 2018. Also, most scientific big data applications need data coupling, i.e., combining data sets physically stored in different facilities. So it is evident that efficient movement of such massive data is not trivial. Youngjae also noted that DOE is planning to enhance their network technology to support a 1 TB/s transfer rate in the near future. But this does not solve the problem because data is still stored on slow devices.

To motivate the research around finding efficient data movement solutions, Youngjae talked about data management at Oak Ridge Laboratory Computing Facility, which runs Titan, the fastest supercomputer in the US. The spider file system based on Lustre provides the petascale PFS for the multiple clusters including Titan. Concurrent data accesses from clusters lead to contentions across multiple network, file system, and storage layers. In such an environment, data movement is usually performed through data transfer nodes (DTN), which access the PFS and send data over the network to the destination's DTN. Because of contentions, PFS can become bottlenecked during data transfers across DTNs. The main problem here is that the difference between the network bandwidth and I/O bandwidth is growing, and PFS's I/O bandwidth is underutilized in the existing schemes of data transfers.

To improve the utilization of PFS's bandwidth on DTN for big data transfers, Youngjae introduced Layout Aware Data Scheduling (LADS) as a solution. He first explained the problem with traditional approaches and used Lustre as the base PFS. He described Lustre's architecture, which contains a metadata server (MDS) and an object storage server (OSS). MDS holds the directory tree and stores metadata about files and does not involve file I/O. On the other hand, OSS manages object storage targets (OSTs), which hold the stripes of file contents and maintains locking of file contents when requested by Lustre clients. In a typical scenario, the client requests file information from MDS, which returns the location of the OSTs holding the file. The client then makes RPC connections to file servers to perform I/O. Youngjae made some key observations about Lustre PFS: it views the file system as single namespace. But the storage is not on a single disk—rather, storage is on multiple disk arrays on multiple servers. PFS is also designed for parallel I/O and traditional data transfer protocols and does not fully utilize inherent parallelism. The main reason behind this is that the traditional file-based approach completely ignores the layout information of the file. For instance, if two threads are working on different files present over the same OST or disk, the threads contend for OST access. In such cases, the thread application runtime increases by 25%. In another case, when multiple threads work on a single file, the parallelism is limited by stripe sizes across OSTs. Multiple threads can contend for different stripes of the same file on the same OST, increasing the runtimes by 50% in some cases. Existing toolkits for bulk data movement (GridFTP, bbcp, RFTP, and SCP) suffer poor performance because of this logical view of files.

LADS uses a physical view of files with which it can understand physical layout of the objects of the file, the storage target holding the objects, and the topology of storage servers and targets. With this knowledge a thread can be scheduled to work on any object on any OST so that we can avoid contention that could have happened in earlier cases. LADS has four design goals: (1) maximized parallelism on multicore CPUs, (2) portability for modern network technologies (using Common Communication Interface (CCI)), (3) leverage parallelism of PFS, and (4) improved hotspot/congestion avoidance, which leads to an end-to-end data transfer optimization reducing the difference between faster network and slower storage. Youngjae then gave a lucid description of the LADS architecture. Data transfers happen between two entities called LADS source and sink. Each of them implements CCI for communication and creates RMA buffers on DRAM. It implements three types of threads: master thread to maintain transfer state, I/O thread for reading/writing data chunks, and comm thread for data transfer across DTNs. Moreover, LADS implements layout-aware, congestion-aware, and NVRAM-buffering algorithms. LADS implements as many OST queues as there are OSTs. A round robin scheduler removes the data chunk from these queues and places it in an RMA buffer while it's not full. When it's full, the I/O threads are put to sleep while the comm thread transfers the contents of the RMA buffer. Once the RMA is available, the I/O threads carry on the leftover I/O and place the chunks in respective queues. I/O congestion control is implemented using a threshold-based reactive algorithm. A threshold value is used to determine the congested servers, and a skip value is used to skip a number of servers. An NVM buffer is also used as extended memory if the RMA is full, which is a typical scenario when the sink is experiencing widespread congestion.

With this overview, Youngjae presented the performance results of LADS. Two Intel Xeon servers were used as DTNs in the testbed and connected with two Fusion-I/O SSDs for NVM and were backed by a Lustre file system over 32 HDDs. An IB QDR 40G network connected the two DTNs. To fairly evaluate the framework, the author increased the message size to make sure the storage bandwidth was not over-provisioned compared to network bandwidth. At 16 KB, the maximum network bandwidth was 3.2 GB/s whereas the I/O bandwidth was 2.3 GB/s. A snapshot of the spider file system revealed that 85% of files were less than 1 MB, and the larger blocks occupied most of the file system space. So for benchmarking, one hundred 1 GB files (big files workload) and ten thousand 1 MB files (small files workload) were used. For the baseline, in the environment without congestion, the transfer rate increased linearly with the number of I/O threads for both workloads, whereas traditional bbcp does not improve with an increase in the number of TCP streams to operate on the same file. This experiment also revealed that LADS moderately uses CPU and memory.

LADS was also evaluated on a storage-congested environment. To simulate congestion, a Linux I/O load generator was used. When a source was congested, a congestion awareness algorithm performed up to 35% better than the baseline configuration without congestion awareness. When sink was congested, the performance impact was significant compared to that of the source congestion. Due to time constraints, the author summarized the remaining experiments. He evaluated the effectiveness of the NVM buffer at source. Throughputs increased with an increase in the size of the available buffer. Actual DTNs at ORNL were also evaluated by transferring data from one cluster with a 20 PB file system to another cluster; LADs showed 6.8 times higher performance compared to bbcp. Youngjae summarized this section and reiterated the effectiveness of layout-aware and congestion-aware models in efficient data transfers. He concluded the talk with his vision to have an optimized virtual path for data transfer from any source to any sink so as to promote collaborative research among organizations.

In the Q&A session, one participant asked the author to compare and contrast the Lustre community's Network Request Scheduler with LADS. Youngjae pointed out that with knowledge of file layout, many smart scheduling schemes could be employed at the application level. The questioner pointed out that the perfectly working environment used in the evaluation does not exist in production environments like that of ORNL, where events like disk failures and Lustre client evictions are common. The questioner wondered whether the authors considered such scenarios in designing LADS. Youngjae mentioned that the current work does not deal with failures.

Having Your Cake and Eating It Too: Jointly Optimal Erasure Codes for I/O, Storage, and Network-Bandwidth

K.V. Rashmi, Preetum Nakkiran, Jingyan Wang, Nihar B. Shah, and Kannan Ramchandran, University of California, Berkeley

K.V. Rashmi began by presenting background for using redundancy in distributed storage environments. Historically, durability and availability of data became essential elements in the design of distributed storage. One of the popular ways to achieve this redundancy is to replicate data across multiple locations. An alternative approach is to employ erasure code to achieve this. It is well known that erasure codes utilize the storage space more efficiently to achieve redundancy when compared to replication. Traditional codes like Reed Solomon provide the maximum possible fault tolerance for the storage overhead used. But the more interesting metric in evaluation of erasure codes is the maintenance costs for the reconstruction of lost data in order to maintain the required level of redundancy. This is quite a frequent operation in distributed systems and becomes heavy in terms of network and I/O. Traditional erasure codes are particularly inefficient in this respect. Both the theory and systems research communities have been working on this problem for quite some time. As a result, a powerful class of erasure coding framework is proposed that optimizes storage and network bandwidth costs of reconstruction. Rashmi extended this class of coding techniques to optimize for I/O as well.

Before delving into the details of this work, Rashmi gave an overview on why traditional codes are inefficient for reconstruction. In a replicated environment, to reconstruct a lost block, one block from a different location has to be transferred, which incurs the cost of I/O and network in transferring the block. In general, one of the popular codes like Reed Solomon takes in k data blocks and generates $n - k$ parity blocks using carefully designed functions for optimal storage and fault tolerance. Now any of the k out of n blocks can be used to reconstruct the missing data. It is clearly evident that the I/O and network costs bump up k times compared to replication cost; for typical variables this cost is increased 10–20x. To address this problem, one of the effective solutions is to use Minimum Storage Regeneration (MSR) codes, which optimize storage and network bandwidth by figuring out the minimum amount of information needed to transfer for reconstruction. The MSR framework, like Reed Solomon, has k data blocks and $n - k$ parity blocks. Under MSR, any block can be reconstructed by connecting to any d ($> k$) helper blocks of remaining blocks and by transferring a small amount of data from each. The total amount of data transferred is significantly smaller compared to Reed Solomon. Although MSR optimizes storage and network bandwidth, it does not consider I/O optimization. In fact I/O is much worse compared to Reed Solomon because the nodes performing reconstruction read the entire block and transfer only a small amount of information. This optimizes bandwidth but performs d full block reads (I/O) whereas Reed Solomon performs only k block reads. In summary, the MSR framework optimizes storage and network but performs much worse in I/O. This work attempts to improve the I/O while retaining the storage and network bandwidth optimization.

Rashmi presented two algorithms that transform MSR codes into codes that provide efficient I/O, storage, and network. The first algorithm transforms MSR codes so that they locally optimize I/O at each of the helper blocks, while the second algorithm builds on top of first to minimize I/O costs globally across all blocks. Rashmi first discussed the kind of performance that can be obtained by applying the transformations. The transformations are applied to a class of practical MSR codes called Product Matrix MSR (PM) codes. PM codes work with storage overhead of less than 2x and provide optimal fault tolerance. They are usually employed in applications that need high fault tolerance. The author implemented both the original PM codes and transformed (PM_RBT) codes in C and evaluated them on an Amazon EC2 cluster, employing Jerasure2 and GF-complete for implementing finite field arithmetic and Reed Solomon. The evaluation was performed for six data blocks with five parity blocks and a block size of 16 MB. The author observed that both the original PM and PM_RBT had approximately 3.27x less data transferred compared to Reed Solomon and emphasized that transformed code retained the bandwidth optimality of the original PM codes. In terms of IOPS consumed during reconstruction, it was evident that PM codes required more IOPS than Reed Solomon. PM_RBT saved up to 5x fewer IOPS than PM and 3x fewer IOPS than Reed Solomon, showing significant performance improvement in terms of I/O. A similar trend was observed for the higher block size of 128 MB. Rashmi next presented I/O completion time for reconstruction. The transformed code (PM_RBT) resulted in 5 to 6 times faster I/O making it significantly faster than Reed Solomon and PM codes.

With performance results in mind, Rashmi described how the transformations work using two algorithms. In MSR, a helper unnecessarily reads a whole block even though it only needs a minor portion of the block; optimally, it reads just the information that's needed. When a helper works in this approach, it's called "reconstruction-by-transfer" (RBT), i.e., it does not do any computation but just reads and transfers. The first algorithm transforms MSR codes to achieve RBT to the maximum extent possible and is applicable to all MSR codes with two properties. First, the function computed at the helper is not dependent on blocks from other helpers (i.e., each block has a predetermined function that aids in reconstruction of another block). The next property deals with independence of these functions at a particular block. Any subset of functions, which produces data of block size, is considered to be independent. The main idea behind this algorithm is to pre-compute and store the results of one such independent set in a block. So now this block can be used for the reconstruction of the blocks corresponding to the functions in the subset. Under MSR, whenever a helper does RBT, it can simply read the data corresponding to a particular function in the block, and hence only a minimum amount of data is read and transferred. But the question of how to choose which functions for which block is dealt with by the second algorithm. Rashmi mentioned that it uses a greedy approach to optimally assign RBT-helpers to minimize I/O cost globally across all the blocks and asked the users to refer to the paper for detailed information on how it's done. She discussed two extreme cases of this algorithm: (1) all the blocks helping all the data blocks to the maximum extent possible (SYS pattern) and (2) all the blocks getting equal treatment and each block helping the following blocks in a cyclic fashion (CYC pattern). Rashmi also evaluated and concluded that the RBT does not affect the decoding speed of PM and is similar to that of Reed Solomon. The encoding speed is slower than that of Reed Solomon but is still practical; she also observed that RBT-SYS has a higher encoding speed than PM. With this, Rashmi summarized the work and restated the benefits brought in by these transformations.

In the Q&A session, one participant was curious about how the compute bandwidth was traded off, because with more helpers the speed is retained but the equations are more complex. Rashmi agreed that there is a compute tradeoff in resource utilization optimization, but it is still practical enough. She reminded the audience that the compute cost during decoding was similar to the cost of RS decoding with two parities. The compute cost is actually not too high because the amount of data churned is much less compared to Reed Solomon. Keith Smith (NetApp) pointed out that the presentation examples used one erasure and wondered about how the efficiency would turn out when there are multiple erasures. Rashmi replied that currently the MSR framework considers optimizing for a single failure and reiterated since only *d* helpers are needed for reconstruction there is room for considering multiple failures in that respect. She also noted that most of the typical scenarios are single failure.

The Fault in Our Stars: Reliability

Summarized by Zhen Cao (zhccao@cs.stonybrook.edu)

Failure-Atomic Updates of Application Data in a Linux File System

Rajat Verma and Anton Ajay Mendez, Hewlett-Packard; Stan Park, Hewlett-Packard Labs; Sandya Mannarswamy, Hewlett-Packard; Terence Kelly and Charles B. Morrey III, Hewlett-Packard Labs

Anton Ajay Mendez presented this paper on behalf of his colleagues in HP Labs, which mainly introduces the design, implementation, and evaluation of failure-atomic application data updates in HP's Advanced File System (AdvFS). Failure-atomic updates would want applications to make their data structures persistent and consistent. Applications would maintain a series of atomic sync points, which provides the ability to revert back to the previous successful sync point in the event of failures. Ajay compared their work with existing mechanisms, including relational databases and key-value stores, and listed some of the previous work and their limitations. Inspired by failure-atomic Msync, their work provides a better and more generalized solution than Msync.

Ajay then came to their solution, which is called O_ATOMIC. It is a flag that can be passed to open() system call, and every subsequent sync would make intervening writes atomic. They also provided another call, syncv, for failure-atomically modifying multiple files. The detailed implementation of their solution leverages a file clone feature, which is a writable snapshot of the file. When a file is opened with O_ATOMIC, a clone of the file is made. When the file is modified the changed blocks are remapped via COW mechanism, while the clone still points to the original blocks. When a sync is called, the old clone is deleted and a new clone is created. Recovery happens lazily on lockup: if a clone exists, delete the file and rename the clone to the original file. Ajay said that their mechanism can be implemented in any file system that supports clones.

To verify the correctness of the mechanism, they injected two types of failure: crash point tests and power cycle tests. Ajay said that test results showed that no corruption happened when files opened with O_ATOMIC. He compared the performance of their implementation against existing key-values stores, and proved it is efficient enough. He also mentioned some caveats of their implementations. Clones introduce fragmentation, but online defragmentation may help here. Multi-process file updates need to be carefully coordinated by the application.

During the questions, Haryadi Gunawi (University of Chicago) asked whether they evaluated how much fragmentation the deployment of their implementation introduced. Ajay replied that they haven't done evaluation on that yet. Justin Paluska (EditShare) asked why multi-process coordination is complicated. Ajay answered because you need to take care of the concurrency issue. What's more, if multiple processes opened the same file, and one process crashed, there is no way to detect it. John Ousterhout (Stanford University) asked about the performance implications of this mechanism, especially when cloning large files. Ajay said if you do the sync immediately after the modification, there is an additional overhead, and it is related to the amount of data changed.

A Tale of Two Erasure Codes in HDFS

Mingyuan Xia, McGill University; Mohit Saxena, Mario Blaum, and David A. Pease, IBM Research Almaden

Mingyuan Xia gave a lively presentation of their new erasure-coded file system, Hadoop Adaptively-Coded Distributed File System (HACFS). He began by showing the fast-growing trend toward global data and laying out the timeline of distributed storage systems. Because of high storage overhead, most systems began using erasure coding instead of the original replication-based methods. Mingyuan gave an overview of erasure coding, using the Reed-Solomon code in Facebook's HDFS as an example. However, erasure coding brings the problems of high degraded read latency and longer reconstruction time. He further explained the two problems in detail. Their goal in this work is to design a technique that can achieve faster recovery as well as lower storage overhead.

Mingyuan then presented the HDFS data access skew. Ten percent of the data gets the majority of the accesses, and 90% of the data is accessed only a few times. Motivated by this finding, they coded read hot files with a fast code with low recovery cost, and coded the majority of the data with a compact code with high storage efficiency. He used the product code family as an example to illustrate this idea. The system will adapt to the workload by converting files between fast and compact codes. Mingyuan described the details of this conversion. When total storage overhead exceeds the storage bound, the system will select files encoded with fast code and upcode them to compact code. Similarly but less likely, when the system is way below the bound, files would be chosen to be downcoded. He provided an example of upcoding operation of the product code family.

Mingyuan then presented the details of environment setting and workloads for evaluation of HACFS. He also formally defined their evaluation metrics: they are degraded read latency, reconstruction time, and storage overhead. He compared HACFS with other distributed storage systems as well as with other erasure coding mechanisms. Mingyuan showed that HACFS always maintains a low storage overhead, while improving the degraded read latency, reconstruction time, and network and disk traffic to a certain extent.

During the questions, Brent Welch (Google) asked whether the codes being compared all have the same redundancy. Mingyuan answered yes. Welch further asked about the data trace, because if we have too much cold data, the recovery of cold data would dominate. Mingyuan answered that in the HDFS data access skew, they showed that the majority of the files are created, read only once, and then stay there. Another researcher asked whether we will lose redundancy by using erasure codes. Mingyuan replied that in three-way replication, replicas are supposed to be placed in different nodes, and it is the same case with erasure codes. So when a single machine fails, no two blocks in one strip will lose at the same time. Rashmi Vinayak (UC Berkeley) asked whether reliability decreases after converting from fast codes to compact codes. Mingyuan answered that they have shown that it is no worse than the three-way replication and is comparable to LRC codes.

How Much Can Data Compressibility Help to Improve NAND Flash Memory Lifetime?

Jiangpeng Li, Kai Zhao, and Xuebin Zhang, Rensselaer Polytechnic Institute; Jun Ma, Shanghai Jiao Tong University; Ming Zhao, Florida International University; Tong Zhang, Rensselaer Polytechnic Institute

Jiangpeng Li began by giving an overview of NAND flash memory. He first explained how NAND flash memory cells gradually wear out with program/erase (P/E) cycling. Methods have been proposed to improve the lifetime of flash memories, including log-structured file system, flash translation layer, error correction coding, and data compression.

Jiangpeng claimed that because of unused space in one NAND flash page and the impact of compression ratio variance, the common sense perception of the quantitative relationship between data compressibility and memory lifetime improvement will not always hold. Moreover, NAND flash memory may further experience content-dependent memory damage. He then showed test results on raw bit error rates (BER) of four different patterns of content to support this. Inspired by this finding, they fill unused space with certain bits so that the number of low-damage patterns is increased and high-damage patterns are reduced. Jiangpeng then proposed implicit data compression as an alternative to complement explicit data compression.

Jiangpeng introduced the damage factor to quantify the impact of different content on memory cell damage. He then derived the mathematical model needed for estimating flash memory lifetime. Simulation results on storage device survival probability when storing different types of files showed the lifetime improved through use of data compression storage techniques. He also discussed the impact of different data compression ratio means and variances on the lifetime gain.

During questions, Nirmal Saxena (Samsung) asked whether the sensitivity in lifetime would change if the wear-leveling algorithm was introduced. Jiangpeng answered that their model can support various kinds of data types. Another researcher asked about the effect of one compressed block being placed on two blocks on the lifetime of flash memory. Jiangpeng replied that in this work they assumed that data are compressed by data sectors. Justin Mazzola Paluska (EditShare) asked how using encrypted file systems would affect the results of this work. Jiangpeng answered that data compression actually would complicate the file system, and in this work they just assumed the granularity of compression is a data sector.

RAIDShield: Characterizing, Monitoring, and Proactively Protecting against Disk Failures

Ao Ma, Fred Douglis, Guanlin Lu, and Darren Sawyer, EMC Corporation; Surendar Chandra and Windsor Hsu, Datrium, Inc.

ACM Test-of-Time Fast Track Award!

Ao Ma gave a lively presentation on disk failure analysis and proactive protection. Ao began by showing that disk failures are commonplace and by giving an overview of RAID. Adding extra redundancy ensures data reliability at the cost of storage efficiency. By analyzing the data collected from one million SATA disks, they revealed that disk failure is predictable, and built RAIDShield, an active defense mechanism, which would reconstruct failing disks before it becomes too late.

Ao then gave the formal definition of a whole-disk failure and showed the statistics summary of their data collection. Disk failure distribution analysis showed that a large fraction of failed drives fail at a similar age. Also, the number of affected disks with sector errors keeps growing, and sector error numbers increase continuously. As a result, passive redundancy is inefficient. Instead, RAIDShield would proactively recognize impending failures and migrate vulnerable data in advance. Using experimental results, Ao showed that reallocated sector (RS) count is a good indicator for failures, while media error count is not. He then characterized its relation with disk failure rate and disk failure time.

Based on previous findings, their single disk proactive protection, called PLATE, uses RS count to predict impending disk failure in advance. Experiment results showed that both the predicted failure and false positive rates decrease as the RS count threshold increases. Ao analyzed the effects of PLATE by comparing the causes of recovery incidents with and without proactive protection, and found that RAID failures were reduced by about 70%. However, PLATE may miss RAID failures caused by multiple less reliable drives, which motivated them to introduce ARMOR, the RAID group proactive protection. Ao then gave an example of how disk group protection works. It calculates the probability of a single disk failure and a vulnerable RAID. Evaluation results also show that ARMOR is an effective methodology to recognize endangered disk groups. In the end, Ao went through some of the related work.

During the questions, John Ousterhout (Stanford University) asked whether the distribution of lifetime of failed drives were percentages of total disks or only of failed disks. Ao answered they are the percentages of failed drives. John followed up about the peak time of failure. Ao replied that they are still not sure why the peak time happens around the third year. Another researcher asked whether all the disks ran for the same span of time. Ao answered yes. They wouldn't replace a disk if there were no errors. A questioner wondered about the zero percentage of failures in the first year. Ao replied that their paper covered six different types of drives, but his presentation only examined the failure patterns of two types. Arkady Kanevsky (Dell) asked whether their model would hold for disks of different types and manufactures and whether their model indicates that we could step back from more complicated protection methods into simpler models. For the first question, Ao replied that now they only considered SATA drives and that it would be interesting to try other types of disks. Regarding the second question, Ao said that they are still polishing their monitor mechanism.

2015 USENIX Research in Linux File and Storage Technologies Summit

February 19, 2015, Santa Clara, CA

Summarized by Rik Farrow

Christoph Hellwig, a Linux I/O developer, opened the workshop by having participants introduce themselves. There were professors and students from Florida International University (FIU), Carnegie Mellon (CMU), Stony Brook University (SBU), Kookmin University (KU), and the University of Wisconsin (UW) present, as well as people from Google, EMC, Red Hat, Parallels, and IBM Research. By the end of the afternoon, one proposal would gain enthusiastic acceptance, while the person who had a proposal accepted in 2014 updated us on his progress.

Attendees had submitted six proposals for discussion, four of them based on FAST '15 papers. Christoph wanted to start working through the proposals immediately, but Erez Zadok asked whether there could first be some discussion of how the Linux kernel submissions process works.

Christoph mentioned that you can use the various kernel mailing lists (see [1] for names), and then displayed a nice diagram for the Linux storage stack [2]. Christoph mentioned different list names, as he pointed to different blocks on the diagram. He used the linux-scsi list most often, but there are other lists, with the kernel list being the most useless because of the amount of list traffic.

Greg Ganger (CMU) asked what Christoph was using for his Non-Volatile Memory (NVM) development. Christoph said that he has an emulator that allows him to play with NVM. Erez pointed out that there is no single way to use NVM, and Christoph responded that some vendor has been using NVM for a while. One way of handling NVM is to mmap() sections of it into user memory. Erez then asked where NVM would fit into the diagram, and Christoph said it was modeled on the direct I/O code (DIO) alongside the Virtual File Systems (VFS) box.

Raju Rangaswami (FIU) asked how the I/O path would be laid out for NVDIMMs (NVM on the memory bus as opposed to PCIe), with the answer also being DIO. James Bottomley (Parallels) said that the diagram is a bit misleading, as all block I/O goes through the block cache, but DIO has its own structure.

Don Porter (SBU) asked about the politics surrounding DIO. Christoph replied that Linus doesn't like it. DIO is the generalization of the classic UNIX raw device. Raw devices bypass the block cache, and you can set a flag turning off the block cache for a device. Database developers and vendors, like Oracle, as well as virtualization developers want to manage block caches for themselves, which is where the interest in raw devices comes from.

Raju moved the discussion back to NVM, asking about Linux developers' view on the future of persistent memory. Christoph answered that what matters is the implementation but not the semantics. Right now, there are two cases: NVM that works like DRAM, and NVM that sits on the PCIe bus. If NVM works like DRAM, it gets treated like DRAM. James pointed out that NVM is not identical to DRAM, because it has slower writes. He also said that we don't have a view—we are looking for a good implementation. Intel has had some interesting failures in this area, and implementation for NVM appears to be an iterative process. Christoph retorted that some NVM is battery-backed DRAM, so it may be identical.

Erez asked about support for Shingled Magnetic Recording (SMR) and zone-based storage devices. Christoph replied that vendors have been pushing SMR on the kernel developers very hard, and they don't like that. James mentioned that they don't have pure SMR drives yet, because if used improperly, the drives' performance would be bad. Christoph described that scenario as host-managed drives, where you manage the drive yourself, and that host-managed SMR will not be in the kernel soon. The kernel can detect these drives, and use the SCSI path to support them, but you need your own code to manage them. Managed drives, what is currently being sold today, handle all the work of SMR themselves, transparently. Especially if you have a window into the drive and a high-speed camera, joked Christoph, referring to a paper that reverse-engineered a managed SMR drive [3]. There is another SMR type, host-aware, where the drive provides some information to the device driver, so you can optimize your drivers to work with it.

Someone thought SMR was an interesting idea but didn't like the interface the vendors were talking about for the host-aware. Christoph agreed, saying it was not a good fit. A vendor could build an in-kernel layer to support host-aware drives, if they were interested. So far, work on host-aware is actually merged into the kernel, but it is like the T10 (object storage standard) code: bit rotting because it is relatively unused.

Remzi Arpaci-Dusseau (UW) asked Christoph about other areas where he wished people were doing more research, and Christoph immediately replied practical cache management algorithms, backed up by James who asked for useful cache work. Christoph elaborated, saying that there has been very little research into multiple stacks of caches. Don summarized these points as practical cache management, algorithms for multiple interacting caches, and asked for more specific examples. Christoph responded that the inode cache is the "mother of all caches," then the kernel caches for data and pages, and the really interesting one, the directory entry (known as dentry) cache (also called dcache). Anything in the inode cache must have a corresponding dentry. When memory pressure forces cache evictions, inodes can't be freed unless there are no dentries referencing them.

Greg asked why dentries were so critical, as they only have to be used for system calls that use pathnames. Christoph replied that certain applications make a lot of use of pathnames, like build farms and unpacking zipped files. James added that most of the standard operational ways of handling files use names; that's why the Postmark benchmark is so important, as most mail servers manipulate many files by name.

Don followed up, asking about interactions with device level caching, and Christoph said they didn't have much data for that. Don observed that they have a lot of problems when memory pressure occurs with determining which cache entries can be thrown away, and throwing away the wrong entry is a problem. Greg added that it's not so much stacked caches as cache interactions that are the real problem.

Christoph named the kernel code used to reclaim memory as the shrinker. The shrinker has a direct hook into every cache, a routine in each cache handler that is supposed to create free space by releasing cache entries. And the space freed needs to be in large blocks that can be passed to the slab allocator. James explained that entangled knowledge, links between caches, caused a lot of problems for the shrinker. Someone wondered why not just use Least Recently Used (LRU) algorithms for shrinking caches. Greg explained that if you remove an inode entry, you also have to remove all the pages and dentries that the inode entry refers to. Christoph agreed, saying that the kernel does use LRU, but if a cache entry has lots of dependencies, it gets put back into the queue of entries, while entries without dependencies get freed. Robert Johnson (SBU) restated this by saying that you can throw away the oldest stuff, but not the oldest stuff that has dependencies, a description that James agreed with.

Don pointed out that what you actually want back are pages. Ted Ts'o (Google) replied that certain objects are more likely to be pinned, so if you want page level LRU to work, directory inodes and dentries need to be on their own pages. Robert asked about page size, which is usually 4K, and James pointed out that Intel doesn't use powers of two for page size, which can either be 4K or 2 MB currently.

Christoph attempted to start the proposal discussions for the fourth time, but we took a short break instead. When we returned, Vasily Tarasov (IBM Research) suggested using a file system instead of caches, but Christoph said that this makes the problems even more complicated. All file systems use the core VFS (Virtual File System) for common services. Vasily pushed his point, but both James and Ted Ts'o said, "No." Ted Ts'o said that items that are referenced must be there and can pin a page. Christoph mentioned that compression might be used on inactive entries, but that would involve following pointers or using table lookups to find entries. The kernel developers had tried different structures, such as trees, but once there is a reference count, there are pointers between entries.

Vasily asked whether FUSE could be used to experiment with caches, but Ted Ts'o responded that FUSE uses caches heavily as a deliberate design choice, making FUSE a poor candidate for experimentation. Christoph suggested writing your own user space VFS, but it would be simpler to work with the kernel. Ted Ts'o then explained that it would actually be easier working within the kernel, as there are very rich debugging tools, much better than in user space. Try using gdb to debug a highly multi-threaded program.

Jun He (UW) asked about conflicts that can cause thrashing. James replied that the kernel is a highly interactive system, and that kernel developers do their best to avoid thrashing. The best way to create thrashing is through memory pressure.

Someone pointed out that Linux is the only OS that uses a dentry cache. Christoph agreed that Linux has a strong dentry cache. Every open file has a filename associated with it, which helps developers with the problem of locking in cross-directory domains. Because of links, a file can have multiple parents, and you could deadlock when manipulating names. This is an area that can be extremely racy. James added that in UNIX, we think of files as names. In Linux we started out with inodes as the primary object. It would be a massive effort to change that. Ted Ts'o also weighed in, saying that this design allows a lot more of the complexity to be pushed up into the VFS layer, and all the details of locking can be down in the core VFS instead of in the underlying file system. Locking is one of the hardest things to do right.

Don asked about the difference between how Linux and BSD perform name lookups. While all systems start by passing the entire path to namei(), BSD and other System V-based UNIX systems process the path, while Linux iterates over each component of the path. Christoph answered that the VFS has to do work for each pathname component anyway. Ted Ts'o added that if the file system is left to figuring out the inode specified by a pathname, the file system has to understand mount points, which can cross file system boundaries.

The Proposals

While all of the previous discussion might appear to be a big distraction, a goal of the Linux developers is to help potential kernel developers become involved. There is an existing culture, as well as process, for making changes to the Linux kernel, and people who wish to make changes need to understand both.

I also asked Ric Wheeler (Red Hat), who started the workshop, but missed this one because he was snowed in near Boston, how he would describe the workshop:

"The purpose of the workshop is to get the Linux kernel community and the FAST file and storage research communities to know each other and our broad portfolio of work. On the FAST side, a large chunk of [research] is Linux kernel based and it is a challenge to know who to talk to in the Linux world and what the Linux community is up to. It is also good to know if the Linux community has already fixed a problem before a poor grad student launches a thesis built around a solved issue."

Proposal 1: Non-Blocking Writes to Files

Daniel Campello (FIU) presented the first proposal, based on his FAST '15 paper [4] that attempts to solve the blocking write problem. When a write occurs that will modify a portion of a block, the process has to sleep until the page is brought into memory. In their research, they store their changes someplace in memory until the page arrives, and then they patch the page before unlocking it. Daniel then asked what the kernel developers thought about this approach.

Christoph started by asking about real life workloads that cause these small writes. Daniel didn't answer this question. Ted Ts'o suggested that they measure with a whole bunch of benchmarks to see whether this is a huge win. Perhaps it would help with BitTorrent, which often receives blocks out-of-order. Daniel stated that they did put traces on writes, they checked for hits or misses, and most applications were weakly consistent. Every time an app wants to write, they recorded the size of the write. They used the SpecSFS benchmarks, and the results seemed to match very well: 30% of writes were not aligned to page boundaries.

Christoph asked how unaligned are the writes, but Ted Ts'o took a different approach: the application is misbehaving. Ted used the example of the BFD Linker, which does unaligned writes, as opposed to the Gold Linker, which doesn't. Use the Gold Linker, said Ted, or fix the app. If they add complexity into the kernel, they have to support it forever. It is easier to fix the few applications that make the kernel more complex for corner cases. Christoph suggested that another idea would be to simplify things: get rid of the struct buffer head, one buffer head per page, and you could have multiple buffer heads per page.

Christoph was a bit more positive overall than Ted, saying that they could use this research to track changes at the byte level to support these types of apps. Daniel said they use a patching mechanism, with the changes hanging from the page headers.

Raju raised another question, about something they noticed during their research, and asked what type of information the developer would like to have about it. Ted suggested tracepoints, which give you the process and username, and said it's not that hard to add a tracepoint to the kernel. Christoph offered to write the tracepoint for them. Raju asked whether this is systemtap, but Ted replied that this is different, and that some subsystems are already heavily tracepointed. The tracepointing system is very efficient by design.

Daniel continued by saying that they collected this data on the file server for both Linux and Windows systems, including a Linux Web server. Daniel thought that their non-blocking writes really solved an important problem. But Ted was not so sure and wanted to understand what was going on at multiple levels of the storage stack to cause this. He worried that there is something important that the developers were not understanding. Daniel mentioned that they ignored files that have holes, and wondered how often files contain holes (regions not yet written to). Christoph answered that it is common for HPC and for VM images but is rare on desktop systems.

Ted continued to worry that they might be missing something important that causes this behavior. Daniel offered to get more information and said that they had avoided collecting too much because of privacy concerns. Raju wondered whether other people had traces that could help. Ted suggested that SQLite could be doing unaligned writes, or MS Word over SMB could be doing it, as well as BitTorrent. Daniel mused that perhaps one of these cases involves a process writing a small file and then going back and overwriting the same page. Greg suggested it happens with large files, when appending without reading first.

Ted said that many kernel developers don't take file servers seriously: cloud servers and personal desktops, yes, but not file servers. There are a set of commercial companies who care about this: those doing small NAS servers. But the kernel developers care more about servers that scale out. Greg imagined a variety of things that could do this, like a logfile that rarely gets written, and its page gets evicted. Daniel said that the Chrome browser actually does this a lot, writes a small file, then overwrites it, every ten minutes. Ted argued that that's a buggy application: Chrome should truncate then overwrite, or write a new file then do an atomic update. The app is doing something that is inherently unsafe.

Erez had a quick follow-up. Key-value stores need to be really efficient, use aligned pages, and use workloads designed to overwrite an entire page, but the kernel still brought in the entire page. James replied that Erez wants trim for the page cache, before Christoph suggested they move on to the next proposal, BetrFS [5].

Proposal 2: BetrFS

Don said that the code they have right now is not ready for prime time, and they wanted to get a sense of what should be done next. Christoph got right to the point: Linux already has over 100 file systems, so you must get people excited about your new file system. Don responded that they like B-epsilon trees, and they plan on writing an article about B-epsilon trees [for *;login:*]. B-epsilon trees are generally an order of magnitude or more faster for inserts than B-trees.

Ted pointed out that kernel developers are very practical. B-epsilon trees are very cool, but what's not immediately obvious is the killer app that needs them. For example, flash file-systems developers have huge commercial imperatives to stabilize those file systems, using 50–200 person years. But for BetrFS, he's not sure why anyone would invest that amount of time. Robert replied that fractal trees [another name for B-epsilon trees] were invented as a backend for MySQL. Using BetrFS with MySQL might also result in 10–30 times the performance.

Christoph suggested that they try to fit their project into an existing file system, then kernel developers could see the advantages by the one-to-one comparison. Ted wondered whether B-epsilon trees would fit well into the BtrFS, but he didn't know about the impedance matching. But the BtrFS folks would like a faster B-tree.

Robert asked whether BtrFS uses a key-value store internally, and Christoph answered that B-trees are used in some places. Robert emphasized that B-epsilon does much faster inserts, and suspected that it will fit into the kernel somewhere. James said that usually they have a problem kernel developers need to solve, and the Stony Brook researchers have a technology looking for a problem. Robert said that they are certain there are problems, and that this technology changes the performance landscape.

Where in file-system land would it be doing massively more updates than queries, wondered Christoph. Robert suggested that every read results in a write because reads mean updating the atime. Someone pointed out that that is why atime updates are disabled by default in Linux. Another person suggested that SQLite does lots of small writes, and Christoph replied that SQLite should have a sensible reimplementation.

Ted asked again about places in the kernel where inserts are more common than queries; he believes this is more common in apps. Erez suggested archival and backups, which Christoph dismissed as write-once-read-never. Greg said that you must read before you can write with backups, while Robert kept pushing, saying there must be a key-value store that will benefit from B-epsilon trees.

Proposal 3: Dcache Lookup

Don said that they had been looking at dcache for a long time. Dcache is used to speed up pathname-to-inode lookups by caching info from past lookups. Don had wondered whether they could speed the process up, and found they could indeed, achieving a 5–26% performance increase.

Don proposed to speed up lookups of non-existent files (which happen, for example, whenever a new file is successfully created) by adding a flag to dcache entries for directories. The flag would indicate that "the dcache contains a complete list of all the entries in this directory." This flag would enable a lookup that fails to find an entry in the dcache to return a definitive "NO." Don described how to initialize this flag as part of a readdir() call and pointed out that whenever an application created a new directory, the flag could be set to TRUE. This could dramatically speed up processes that create an entire directory hierarchy, such as git checkout or untar. Currently, without this optimization, the kernel queries the underlying FS for each file and directory created, even though the parent directory was just created a moment earlier and the kernel has a full cache of its contents. Then Don asked whether the developers would be open to trying new lookup algorithms.

Christoph said simply: send your code to Linus now. Ted said that Linus is really interested in improving path lookup and complains when file stats are slow. Don said that they measure a 26% improvement in getting file status. Christoph repeated, send it to Linus, although Linus will rewrite whatever you send him. Don said he doesn't care, and pointed out that they were compatible with SELinux. Christoph asked whether they still have slowpath, and Don said they do, but worried about their patch being slower when there is a rename in an early directory in a long pathname. Greg said that renames near the beginning of a path don't happen often enough to optimize for, and Ted suggested including a version with the cached info, so it could be invalidated if needed. Christoph felt some concern about getting the readdir() case right, but that the mkdir() case seemed simple enough and gave a path for incremental deployment.

Robert presented the second part of their proposal, to speed up readdir(). They want to keep negative results and add a bloom filter. The bloom filter would be kept when memory pressure results in freeing cached dentries. Ted worried about the memory used, and Christoph said this could be a good research paper.

Proposal 4: Chopper

Ted asked for a recap from the lead author of Chopper [6]. Jun He (UW) said that they had tried their favorite workload and found some problems in the block allocator. They wanted to search the design space for file systems and needed a tool to help with experimental design. They developed Chopper as that tool and uncovered four different issues with ext4.

The first issue involved scheduler dependencies in ext4, when many small files were created in a single directory. Ted asked about the real world scenario where this happens; perhaps if the app tried slow writes using multiple CPUs.

Jun plunged on with the details of their second finding, when four threads are writing to one file. Again, Ted interrupted Jun, saying that they must have delayed allocation disabled, and that examples must be real world apps, not synthetic tests. Ted went on to say that he found their testing framework, Chopper, interesting and that perhaps there were other metrics he'd like to test, such as the average size of free extents. He worried not just about the goodness of written files, but also the goodness of the free list. Another issue would be a change that helps metric A but causes regression in metric B.

Jun never got through all four of their findings. Instead, we broke for coffee, with just 90 minutes remaining in the workshop.

Proposal 5: A New Form of Storage Virtualization

Zev Weiss (UW) described the work in their paper [7], which allows adding features from more advanced file systems, like ZFS and BtrFS, to ext4 and XFS. The idea was to work at the block layer to add features (e.g., snapshots, copy-on-write) transparently to existing file systems. Christoph said that if the researchers could take this further, it could be helpful and very interesting. Ted said that he had written up a proposal for a device mapper for a block, and that there were interesting things that could be done at the device mapper layer. You do need to have some communication between the device mapper and the block layer, but this would be interesting to explore.

Don asked about the standard for getting a device mapper into the kernel; Christoph said this can be done, and Vasily has done this.

Proposal 6: Deduper

Vasily Tarasov (IBM Research) presented his idea for an open source, block-level deduper, at the 2014 Linux FAST workshop. Vasily worked with Christoph, who helped him with his patch set and with submitting the patch set to the device mapper list. They immediately got back the response that they needed more comments in their patch set. In July 2014, Vasily presented a paper [8] on a device mapper deduper. In August 2014, they submitted another patch set, and got some attention, but not a lot of responses. In January 2015, one developer became devoted to this module, working to fix things, and in that respect the process has been good. The amount of work required to keep the process of getting the patch into upstream has varied from month to month.

Christoph asked Erez about the Stony Brook students' poster about dedupe. Erez said that not all data blocks are created equally, that some are metadata. If you are submitting metadata, you want to set a flag for "don't dedupe." A metadata hint was what they needed, and they found that such a flag was there. Christoph said that hint was there for tracing, and could be very useful. Ted asked which file system has Sonam Mandal (not present) worked on? Erez replied that she had worked on all the exts. Ted said that he thought ext4 was well marked, and Erez replied that they are also looking into pushing hints further up the storage stack for temp files and for files that will be encrypted.

Ted said they are interested in discovering which hints have the highest value so are worth implementing. You want to set these hints in the inode, not in the page cache. Christoph worried about trusting users with these hints, as they could be abused or simply used poorly. Erez wanted to put the ability into libc, and Ted thought that such hints would be useful for compiler temp files.

Erez asked about the existence of a metadata flag, and Christoph said it was only used for tracing. That flag used to be in the I/O scheduler, and is still present in ext2 and ext4. Ted said the flag is there for the commit block, and nothing can go out until that commit block has been committed. Erez joked that they didn't want to dedupe the duplicate superblocks, and people laughed. They treated these duplicates as metadata, and wondered whether they should submit patches that do that. Christoph replied, "Sure."

Ted continued the discussion by focusing on priority inheritance in a scenario involving a low priority cgroup process wanting to read a directory, locking it, but being unable to read it, and then a cluster manager needs that block but can't read it. Raju said they found this problem in their work on non-blocking writes, but didn't fix this behavior. Ted said that read-ahead has a low priority, and if you have a process blocked because the process needs that block, that really causes a problem. The right answer is that we need to have better visibility. And the real problem is that these problems have been hiding in the underbrush for years.

Raju asked whether this was a significant enough problem for people to research. Ted replied that if this is a problem on Android, but fixing it would break big servers with high performance I/O, that would cause problems. They won't risk server performance to make Android faster. Samsung can apply that patch locally [9] for their handsets, but we can't push it upstream.

Zev asked what type of interface was imagined for things like block remapping. Christoph replied that they have thought about operations for block devices for a long time. Zev then asked about stable pages, where the page is locked. Ted said they just make a copy of the block as a hack, and that they just worked around it, the copy worked acceptably, but the patch cannot be pushed upstream. James said he has seen a lot of email complaining about the problem. Ted replied that they could send out the patch to see whether there is any interest.

Discussion

We had finished with the formal proposals by 5:15, and Christoph opened the floor to general discussions.

Dungyun Shin (KU) wants to measure times in the local I/O stack performance so he can then learn which layer is causing a problem. If he knows where the problem is, he knows what needs to be optimized. Dungyun asked for feedback on which layer to focus on. Dungyun is currently working on the block I/O layer and the VFS layer.

Christoph said there are two parts to this question. You can get perfect info using blocktrace, and there is a tool there for measuring latency. There are also tools at the system call level, but the problem is combining the two.

Ted said there has been recent related work at Google. They were interested in long tail latency—2, 3, 4 nines latency—and they measured worst-case times, and didn't try cross-correlating. They just wanted to find where the long tail was happening. For them, CFQ (Completely Fair Queueing) was completely out to lunch, or in the hardware, the drive was going out to lunch for hundreds of milliseconds. Rather than trying to cross-correlate all the way down, it was easiest to measure the long tail.

Dungyun asked whether it is a good strategy to correlate interrupt times, since some benchmarks running on Android are very variable, not consistent. They can't trust the research because of this. Dungyun tried experiments to eliminate the time spent I/O handling, and then he got more consistent results.

Ted said he doesn't have a high opinion of many of those benchmark scores—do they actually reflect what the user will see? Measuring how long it takes to start the Facebook app, let's measure that because that app is really big. Handset vendors actually check the application ID so the CPU benchmarking scores can be artificially high. Those scores are there for "benchmarketing." Facebook reads 100 MBs before it displays a pixel. If you want to find out what is actually causing the delay—memory pressure, locking—measure latencies at different levels of the I/O stack. But then, why are you optimizing this?

Don asked about the developers' perspective and the application developers' perspective. Ted countered by asking whether this is the application you really care about? What is the real world case where we are not optimized for this area? If he decided to fix this in ext4, but people who run into this are using xfs, why should he fix this? When thinking about writing papers, pick something that will be high impact—that is, something that will affect lots of users. That's why Ted liked the Quasi I/O paper [9]. But does three seconds instead of one second really matter to the handset user?

The meeting broke up just before 6 p.m., because Google needed to secure the room they had loaned us.

References

[1] Info for Linux developers mailing list: http://vger.kernel.org/vger-lists.html.

[2] Linux Storage Stack Diagram, Werner Fischer: https://www.thomas-krenn.com/en/wiki/Linux_Storage_Stack_Diagram.

[3] A. Aghayev and P. Desnoyers, "Skylight—A Window on Shingled Disk Operation," FAST '15: https://www.usenix.org/conference/fast15/technical-sessions/presentation/aghayev.

[4] D. Campello, H. Lopez, L. Useche, R. Koller, R. Rangaswami, "Non-Blocking Writes to Files," FAST '15: https://www.usenix.org/conference/fast15/technical-sessions/presentation/campello.

[5] W. Jannen, J. Yuan, Y. Zhan, A. Akshintala, J. Esmet, Y. Jiao, A. Mittal, P. Pandey, P. Reddy, L. Walsh, M. Bender, M. Farach-Colton, R. Johnson, B. C. Kuszmaul, D. E. Porter, "BetrFS: A Right-Optimized Write-Optimized File System," FAST '15: https://www.usenix.org/conference/fast15/technical-sessions/presentation/jannen.

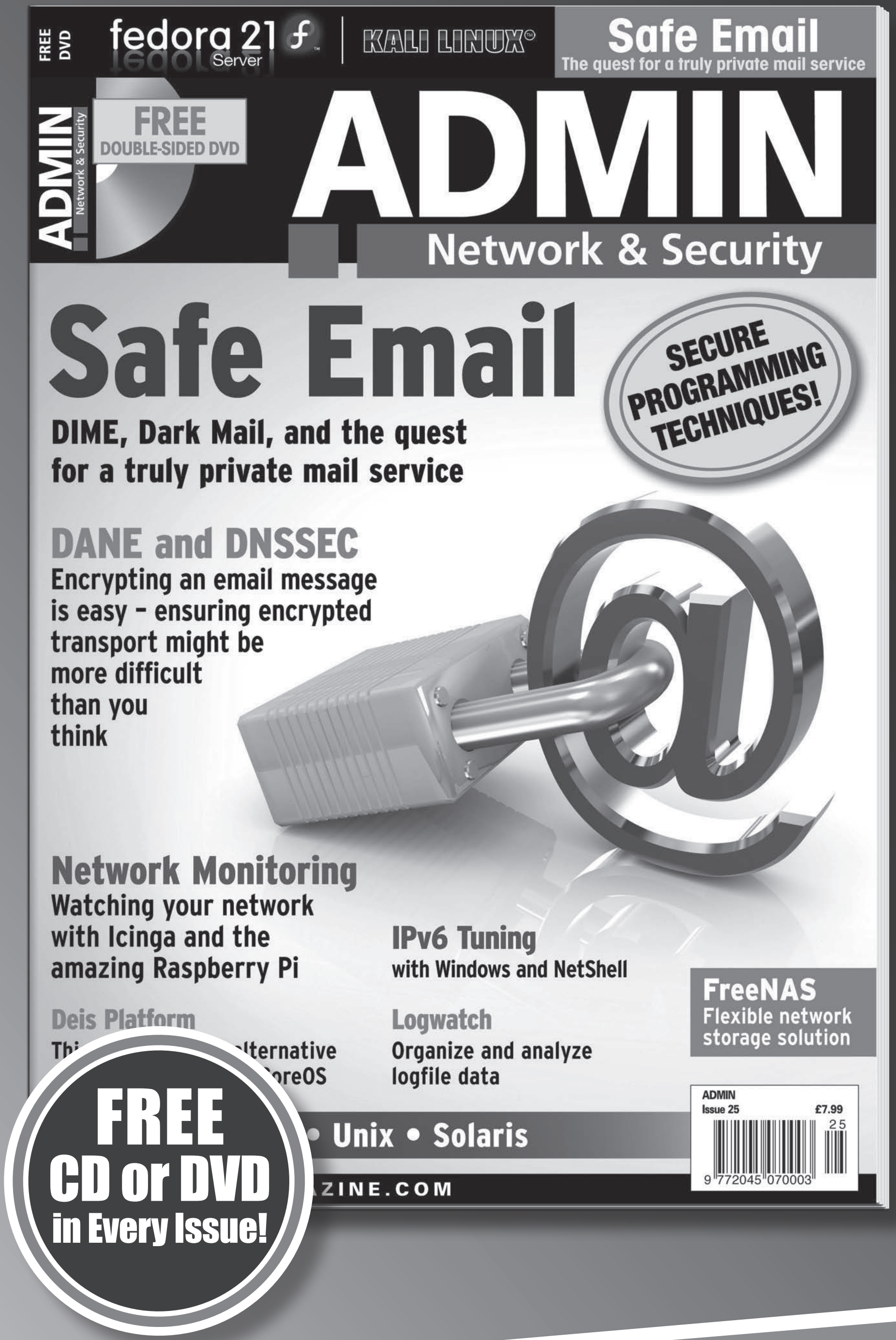
[6] J. He, D. Nguyen, A. Arpaci-Dusseau, and R. Arpaci-Dusseau, "Reducing File System Tail Latencies with Chopper," FAST '15: https://www.usenix.org/conference/fast15/technical-sessions/presentation/he.

[7] Z. Weiss, S. Subramanian, S. Sundararaman, N. Talagala, A. Arpaci-Dusseau, and R. Arpaci-Dusseau; "ANViL: Advanced Virtualization for Modern Non-Volatile Memory Devices," FAST '15: https://www.usenix.org/conference/fast15/technical-sessions/presentation/weiss.

[8] V. Tarasov, D. Jain, G. Kuenning, S. Mandal, K. Palanisami, P. Shilane, S. Trehan, and E. Zadok, "Dmdedup: Device Mapper Target for Data Deduplication," OLS 2014: https://www.kernel.org/doc/ols/2014/ols2014-tarasov.pdf.

[9] D. Jeong, Y. Lee, and J. Kim, "Boosting Quasi-Asynchronous I/O for Better Responsiveness in Mobile Devices," FAST '15: https://www.usenix.org/conference/fast15/technical-sessions/presentation/jeong.

REAL SOLUTIONS FOR REAL NETWORKS



FREE DVD
fedora 21 Server
KALI LINUX
Safe Email
The quest for a truly private mail service
ADMIN
Network & Security
FREE DOUBLE-SIDED DVD
Safe Email
DIME, Dark Mail, and the quest for a truly private mail service
SECURE PROGRAMMING TECHNIQUES!
DANE and DNSSEC
Encrypting an email message is easy – ensuring encrypted transport might be more difficult than you think
Network Monitoring
Watching your network with Icinga and the amazing Raspberry Pi
IPv6 Tuning
with Windows and NetShell
Deis Platform
Logwatch
Organize and analyze logfile data
FreeNAS
Flexible network storage solution
Unix • Solaris
FREE CD or DVD in Every Issue!

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

Learn the latest techniques for better:

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

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

6 issues per year!

ORDER ONLINE AT: shop.linuxnewmedia.com

usenix
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

USENIX Association
2560 Ninth Street, Suite 215
Berkeley, CA 94710

POSTMASTER
Send Address Changes to *;login:*
2560 Ninth Street, Suite 215
Berkeley, CA 94710

PERIODICALS POSTAGE
PAID
AT BERKELEY, CALIFORNIA
AND ADDITIONAL OFFICES

usenix 40TH ANNIVERSARY

24th USENIX Security Symposium

Plus the Co-located Workshops

AUGUST 10–14, 2015 • WASHINGTON, D.C.

usenix
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

The 24th USENIX Security Symposium will bring together researchers, practitioners, system administrators, system programmers, and others interested in the latest advances in the security and privacy of computer systems and networks. USENIX Security '15 will be held August 12–14, 2015, in Washington, D.C.

The Symposium will span three days, with a technical program including refereed papers, invited talks, posters, panel discussions, and Birds-of-a-Feather sessions. The following co-located events will precede the Symposium on August 10 and 11:

WOOT '15: 9th USENIX Workshop on Offensive Technologies
Monday–Tuesday, August 10–11, 2015

CSET '15: 8th Workshop on Cyber Security Experimentation and Test
Monday, August 10, 2015

FOCI '15: 5th USENIX Workshop on Free and Open Communications on the Internet
Monday, August 10, 2015

HealthTech '15: 2015 USENIX Summit on Health Information Technologies
Safety, Security, Privacy, and Interoperability of Health Information Technologies
Monday, August 10, 2015

3GSE '15: 2015 USENIX Summit on Gaming, Games, and Gamification in Security Education
Tuesday, August 11, 2015

HotSec '15: 2015 USENIX Summit on Hot Topics in Security
Tuesday, August 11, 2015

JETS '15: 2015 USENIX Journal of Election Technology and Systems Workshop
(Formerly EVT/WOTE)
Tuesday, August 11, 2015

Register Today!
www.usenix.org/sec15